PLANAR VEHICLE TRACKING USING A MONOCULAR BASED MULTIPLE CAMERA VISUAL POSITION SYSTEM

By

ANTHONY R. HINSON

A THESIS PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTERS OF SCIENCE

UNIVERSITY OF FLORIDA

2003

I’d like to dedicate this thesis to my girlfriend Elisabeth who has stuck by me during the good times, the bad times, and all those in-between.

ACKNOWLEDGMENTS

The process of earning my master's has been a lengthy process, in which many people have offered invaluable assistance. I extend my thanks to all those who stood by me and encouraged me during these last few years. First and foremost, I would like to thank Dr. Carl Crane for his patience with me and for letting me determine the direction of my research. I would like to thank my mother for the endless encouragement, optimism, and, of course, the financial contributions that have kept me in college all these years. I would like to thank my good friend Marion Douglas, who has selflessly allowed me to use his apartment like a hotel during the final, nomadic phase of my research. I would also like to thank several of my colleagues and friends Sai Yeluri, Arfath Pasha, Jose Santiago, and Peter Vinch for their indispensable academic advice and insight. Finally, I would like to thank all of those not explicitly mentioned here who have aided my intellectual and social growth throughout my academic career.

TABLE OF CONTENTS

Page

LIST OF TABLES

LIST OF FIGURES

Abstract of Thesis Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Master of Science

PLANAR VEHICLE TRACKING USING A MONOCULAR BASED
MULTIPLE CAMERA VISUAL POSITION SYSTEM

By

Anthony R. Hinson

August 2003

Chair: Dr. Carl D. Crane III
Major Department: Mechanical and Aerospace Engineering

Vehicle tracking is an essential component to automating unmanned vehicles. Whether controlling a vehicle via tele-remote operation or allowing a vehicle to run autonomously, the global position of the vehicle is always important.

A new position system was proposed that would not only provide highly accurate tracking information for an indoor ground vehicle, but also would be inexpensive and non-intrusive to the vehicle or the environment in which it is located. A visual-based tracking system requires as little as one camera and one computer to give vehicle position information. A vehicle simply needs the addition of two tracking features on its roof to allow the software to find it in the image.

The position system uses the intrinsic and extrinsic properties of the camera to create a series of vectors (one for each pixel) extending from the camera outward in space. The object seen in a pixel intersects the matching vector at an unknown location along the vector in space. By assuming each vector intersects a known plane, 3D range

information can be determined for each pixel in the image. These spatial calculations can be done easily with projective geometry for each pixel and saved into a look-up table. This table contains the physical location that each pixel represents in world coordinates.

To get vehicle position, the tracking features on top of the vehicle must be searched for in every frame. This is done using several image processing methods including color ranging and Gaussian based models. Another method, color direction, was developed solely for this research. This method involves calculating normalized direction vectors based on the color's location in RGB space. Searching for colors with similar vectors will find all shades of a particular color regardless of lighting conditions. Once the features are found, the pixel closest to each feature's centroid is determined and used to look up the location of that feature in world coordinates. The locations of these features are then used to define the vehicle's orientation vector and position in world coordinates.

This method was first tested on simulated 3d environments built in 3d Studio Max. The calculated position of the vehicle in relation to the actual position yielded less than 1% error on average. The program was then tested on actual vehicles in accurately measured environments. The results from these empirical tests were very good despite some hardware difficulties. The first experimental test results gave vehicle position within 2%-3% on average. The second experimental test exhibited some problems in camera modeling but still displayed errors less than 4%.

It can be concluded that planar visual positioning is an inexpensive and accurate vehicle tracking method that is capable of use with autonomous indoor ground vehicles.

CHAPTER 1
BACKGROUND

Planar visual positioning is not necessarily a new concept, but more of a revamp of existing ideas. Vision has been used for years for tracking people and objects in a variety of circumstances. This research entails creating a similar type system, but for the specific usage of indoor ground vehicles. But, before a new position system can be created, existing ideas and concepts need to be examined.

Existing Position Systems

Position systems come in an abundance of varieties each with its positive and negative points. There are far too many position and navigation systems to discuss each one. Instead, a few of the more general type navigation systems are discussed in this section.

Dead Reckoning

Dead reckoning is a positioning method that involves measuring the heading and direction over time to calculate a vehicle's position. This type of positioning is simple and easy to implement, but the results drift over time. Errors in the sensor measurements accrue leading degraded vehicle position [Wit96]. No external references are used in dead reckoning, so error accumulation is unavoidable.

Dead reckoning is typically done on ground vehicles with wheel encoders and a digital compass. The wheel encoders give the distance traveled by the vehicle and the digital compass gives the heading. These two systems working together can provide general detail about position and orientation. This method of positioning is relatively

inexpensive to implement, but can only be used if the position information is not critical for the operation.

Inertial Based Positioning

Inertial based position systems are a type of dead reckoning position system. Inertial measurement units, or IMUs, are widely used on unmanned space vehicles and satellites. In space, there are few objects to use as reference points for navigation and everything is in a state of constant motion, so IMUs are well suited for these types of vehicles.

To fully determine a vehicle's position, the IMU measures the acceleration in the three principal directions (with respect to the unit) and the three rotational angles of the unit with respect to the environment. The acceleration is measured by three or more accelerometers. Accelerometers can be thought of as mass-spring systems that can only move in one direction. As the acceleration increases, the mass pulls on the spring to a displacement proportional to the acceleration. Modern accelerometers are much more complicated than this, but the concept remains the same. The second integral is taken of the acceleration value in each direction to give the distance traveled in those same directions. These principal distance values give the approximate distance traveled by the IMU in its coordinate system.

The second set of data an IMU gives is the rotational displacement. Older IMUs in the mid 20th century used precision machined, rotating spheres to measure the angular displacement of the IMU. Newer systems typically use an optical version of the gyro called a ring-laser gyro (RLG). A ring laser gyro emits a laser, which is split into two beams in its sensitive direction. One beam travels clockwise and the other counter-clockwise. The two beams are then recombined and viewed by a sensor. When no

rotation is present, the lasers will have the same frequency when striking the collector. If rotation has occurred, the Doppler Effect will cause the wavelength of the beams to change. When the beams are recombined, there will be destructive interference between the two which can be read by the collector [FAS03]. This interference is related to the phase shift of the beams and subsequently to rotational rate. This value can then be integrated to determine the IMU's angular position. RLGs are extremely accurate, typically measuring angles in the arc-second range.

The combination of the acceleration data and the rotational information allows for very accurate dead reckoning position measurement. It is important to note that IMUs give the net change in position, not the position itself. To get position coordinates requires that a starting position of the IMU be known.

The main problem with an IMU is the price. The standard, high accuracy IMU runs in the range of $80,000 to $100,000. The prices can go up significantly for IMUs that have redundant systems, increased accuracy, specialized capabilities, etc. The steep price tag for these systems leaves them best suited for military and government funded operations.

Global Positioning System

GPS is a position and navigation system that has been growing in popularity over the last several years. Since the degradation capabilities, known as selective availability, were deactivated in May of 2000 the use of GPS has skyrocketed. The original commercial accuracy of GPS, with selective availability enabled, was in the range of 100 meters. After the deactivation of selective availability, the commercial GPS accuracy jumped to around 10 to 20 meters [Tri03].

With the increased accuracy, this technology has become a popular type of navigation technology worldwide. GPS is used in a variety of markets from personal course-plotting devices to navigation for commercial airliners [Cro98]. This increase in the market has led to a host of new devices and a general decrease in the prices of GPS systems.

Currently, GPS is a powerful navigation system that can be accurate to a few meters at practically any location world wide [Ndi98]. To work well, a GPS navigation device must have a clear line of sight to at least three of the GPS satellites in orbit. The accuracy of a GPS device increases when more satellites are found through redundant calculations. Because of this need for multiple satellites, GPS does not work well indoors or in heavy foliage.

These are just a few of the countless position and navigation systems available today. Some navigation systems are intended for broad range usage, but most are designed for use in very specific environments. Unsurprisingly, many of the navigation systems are developed for military related applications. But as industry becomes more automated, the need for commercial navigation systems is ever increasing.

While most of the personal navigation needs are being met by GPS, the industrial sector is often forced to design its own navigation systems. This is where the planar visual positioning research comes in. This type of position and navigation system has been created with the specific intention of use in an industrial environment, while keeping the small business in mind. Hopes are that if developed for commercial use, this system could be easily and inexpensively installed and serve as an accurate, reliable navigation system for industrial type applications.

Planar Visual Positioning

The concept of planar visual positioning was conceptualized by accident while attempting to visually detect road lines. The goal for the road line detection is to relate the locations of the lines to the position of the camera so that the vehicle can stay on the road. As long as a few basic camera properties are known, it is fairly trivial to determine the position of the road lines with respect to the camera. While attempting to accomplish this task, the prospect of using this method to track objects other than road lines emerged.

Indoor ground vehicles were the perfect candidates for tracking with a planar vision position system. First and foremost, the driving surface indoors is typically much flatter than outdoor settings. Second, environmental conditions can be regulated indoors which allows for better testing conditions. Third, outfitting a warehouse or room with cameras and computer equipment is much easier than outdoors. Therefore, a warehouse was decided to be the test environment for this position system.

The concept is simple: take an existing warehouse, survey its dimensions, and install several video cameras and a computer. Once these things are in place, any vehicle with a set of distinguishable features can have its position tracked by this system. All that is required is some simple geometric conversions to create an image data tracking grid from each camera. Then using some standard image processing techniques, the vehicle's location can be found in the video from each camera and related to corresponding tracking grids. The vehicle's position is then found by simply combining the information from all the separate cameras and tracking grids. Specific details on how this is done span the next four chapters.

Image File Formats

An essential portion of visual positioning is image processing. While the main goal of this position system is to successfully track a vehicle with real-time imaging, a real-time video stream is not always available. For the development of this research, it was decided that the use of previously captured video would be more practical than trying to always use real-time images. Position system concepts and algorithms were tested on both still and moving images. Use of these picture and video files requires basic knowledge of the file formatting. This section gives brief explanations of the file formats used, only discussing information that is necessary for effective use inside the position system.

Still Image File Formats

Still pictures were used throughout this research for calibration and simple correlation between world objects and their associated image representations. It is much easier to compare and contrast details in a single image than to try to do this for a number of images. The first goal of this research, therefore, was to successfully process still images.

Uncompressed Windows bitmap

In the early stages of this research, the image processing software was developed under the Microsoft Windows XP® platform; therefore, the Windows bitmap seemed to be the simplest image format to use for still images. It was decided that despite the drawbacks resulting from using uncompressed images, encoding and decoding compressed images was beyond the scope of this research. The extra time needed to write or adapt a JPEG or GIF decoder was unnecessary.

The data format of a Windows bitmap is uncomplicated but not necessarily intuitive. There are three basic parts to a Windows bitmap: the header, the color palette, and the pixel data. The bitmap's header occupies the first 54-bytes of the image. The header is broken down into two data structures: the BITMAPFILEHEADER and BITMAPINFOHEADER. The bitmap header format is shown in Figure 1.

Much of the data in the bitmap's header relates to lower bit-rate images that are not discussed here. Detailed information about the bitmap header variables is shown in Table 1 [Del02]. The color palette is a series of data structures in the file that convert lower bit-rate color information to 24-bit colors. The color palette is only present on bitmaps that are less than 24-bit. Only 24-bit color images are used in this research so the color palette is only discussed briefly.

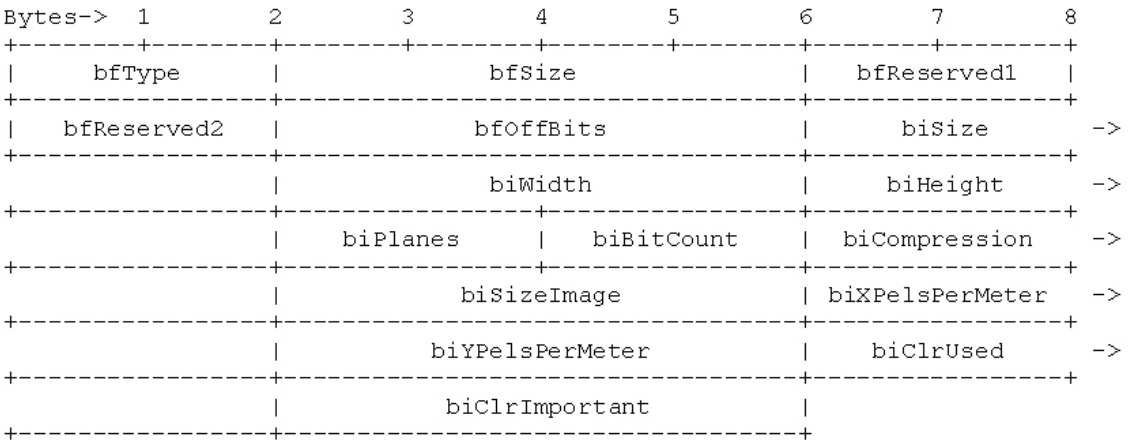

Figure 1: Windows Bitmap Header Data Layout

Table 1: Bitmap File Header Information

Data Descriptor	Req'd for 24-bit BMP Variable Type/Size Offset (bytes) Structure Membership	Description
bfType	Yes WORD (2bytes) 0x00 BITMAPFILEHEADER	These first two bytes of the bitmap are the file signature for a Windows bitmap. To be a properly defined Windows bitmap, this value must be equal to 0x4d42 which, when converted to alphanumeric characters represents 'BM'.
bfSize	No DWORD (4bytes) 0x02 BITMAPFILEHEADER	This is the overall size of the bitmap in bytes. This data can be unreliable, so caution should be used when referencing this value.
bfReserved1	Yes WORD (2bytes) 0x06 BITMAPFILEHEADER	bfReserved1 and bfReserved2 are reserved spaces that must be set to zero.
bfReserved2	Yes WORD (2bytes) 0x08 BITMAPFILEHEADER	See bfReserved1.

Table 1. Continued

Data Descriptor	Req'd for 24-bit BMP Variable Type/Size Offset (bytes) Structure Membership	Description
bfOffBits	Yes DWORD (4bytes) 0x0a BITMAPFILEHEADER	This value is the number of bytes that the image pixel data is offset from the start of the file. Even though the structure name of this variable is bfOffBits the data is actually in bytes. For a full color image, typical of this research, this value will be equal to 54. For an 8-bit, paletted bitmap this value should be 1078.
biSize	Yes DWORD (4bytes) 0x0e BITMAPINFOHEADER	This variable represents the size of the BITMAPINFOHEADER structure in bytes. This value should usually be 40.
biWidth	Yes DWORD (4bytes) 0x12 BITMAPINFOHEADER	This is the width of the image in pixels.
biHeight	Yes DWORD (4bytes) 0x16 BITMAPINFOHEADER	This is the height of the image in pixels
biPlanes	Yes WORD (2bytes) 0x1a BITMAPINFOHEADER	Number of image planes in the bitmap. Multiple planes are rarely used in bitmaps, so this value should typically be set to 1.

Table 1. Continued

Data Descriptor	Req'd for 24-bit BMP Variable Type/Size Offset (bytes) Structure Membership	Description
biBitCount	Yes WORD (2bytes) 0x1c BITMAPINFOHEADER	This variable is the number of bits required to represent the color of the pixel. This value is also known as the color depth. This value can be: 1 – for monochromatic bitmaps, 4 – for 16 color bitmaps, 8 – for 256 color bitmaps, 16 – for full color bitmaps (65,536 colors), or 24 – for true color bitmaps (16 million colors). 24-bit images have the pixel's color information completely described by the pixel data. Lower bit-rate images must be paletted. Color palettes will be discussed later in this section.
biCompression	Yes DWORD (4bytes) 0x1e BITMAPINFOHEADER	This value tells the type of compression used on the bitmap. This variable can be one of three values: 0 – for an uncompressed bitmap, 1 – for a RLE-8 compressed bitmap, or 2 – for a RLE-4 compressed bitmap.
biSizeImage	No DWORD (4bytes) 0x22 BITMAPINFOHEADER	This is the size of the pixel data including padding (in bytes). Data padding will be discussed later in this section.
biXPelsPerMeter	No DWORD (4bytes) 0x26 BITMAPINFOHEADER	biXPelsPerMeter and biYPelsPerMeter are the width and height of the image in pixels per meter. These values are not used for standard resolutions and can be set to zero.

Table 1. Continued

Data Descriptor	Req'd for 24-bit BMP Variable Type/Size Offset (bytes) Structure Membership	Description
biYPelsPerMeter	No DWORD (4bytes) 0x2a BITMAPINFOHEADER	See biYPelsPerMeter.
biClrUsed	No DWORD (4bytes) 0x2e BITMAPINFOHEADER	This variable represents the number of color palette colors used in the image. This value is unimportant for full color images.
biClrImportant	No DWORD (4bytes) 0x32 BITMAPINFOHEADER	This value tells which colors are most important in color paletted images. This information can be used to optimize screen display speeds.

The header information for every Windows bitmap should be in this format. The information that is defined will vary from application to application, but the general format holds true. Images with a color depth of less than 24-bit will have color palette information located between the bitmap header and the pixel data. These color palettes are comprised of *RGB-Quads*, which relate the paletted colors to actual 24-bit screen colors. There must be one quad defined for every possible color in the image, so the number of quads will depend on the depth of the color. A 1-bit image will have only two quads. One quad will represent the screen color of a pixel with the value of zero and the

other quad will give the screen color of a pixel with a value of one. 4-bit images will have 16 quads, 8-bit images will have 256, and so on.

The final part of the bitmap image is the actual pixel data. The formatting of the pixel data is completely different for each type of compression and color depth, but only 24-bit images will be discussed. Pixel data starts at the bottom-left corner of the image and scan lines run left to right. The scan lines are read from bottom to top, so that the last pixel in the file will be the top-right corner. The individual color channel values of each pixel are arranged in with blue first, green second, and red last. Each color component is an 8-bit unsigned integer value that can range from 0 to 255.

Computers today are optimized to process data in 32-bit chunks. For optimization reasons, bitmaps are written so that scan lines always end on a 32-bit boundary. If the total size of a bitmap scan line is not a multiple of 32, the end of the scan line must be padded with zero bits until a 32-bit boundary is reached. For 24-bit images, scan lines are going to be padded when the width of the image is not a multiple of four. The height of the image does not factor into scan line padding.

Portable pixmap file

The portable pixmap, or PPM, is one of the simplest image formats available. PPMs are rare in a Windows platform, but are seen more commonly under Linux and Unix. PPMs are convenient to work with because of their extreme simplicity, but offer nothing in the way of compression and can become quite large. On the other hand, the formatting of PPMs is much more intuitive than the Windows bitmap. Pixel data in a PPM reads like text, from left to right and top to bottom. Color data is listed for each pixel with the red first, green second, and blue last. Two different varieties of PPMs exist: the older P3 format and the newer P6 format.

The P3 is truly the low end of image formats. The entire file is in ASCII, which makes viewing the pixel integer values quite simple, but the file sizes are massive and the slow to display. The header information in a P3 is limited to the horizontal and vertical resolution of the image and the maximum possible value of the pixel components. This information is all in ASCII at the beginning of the file separated by white spaces and carriage returns. The P3 header should look like:

'P3'

<*HorzResolution*> <*VertResolution*>

<*MaxVal*>

Where the characters 'P3' denote the PPM file format, and the horizontal and vertical resolution values are in terms of pixels [Pos02]. The *MaxVal* value is the maximum integer value that any pixel component may have. The value of each pixel is the result of integer of the red, green, and blue pixel components divided by the *MaxVal* value. This makes the conversion from raw data to an image very convenient. The pixel data in the P3 is formatted like the header information, a series of values separated by white spaces.

The disadvantage of the P3 PPM is its excessive size. To get a perspective of how much larger the P3 PPMs are than standard bitmaps, look at the information needed to describe one pixel. For a binary bitmap, a pixel can be completely described in three bytes, one byte for red, one byte for green, and one byte for blue. For the ASCII representation of the same pixel, a data string like this is needed: '255 255 255 ' to describe the same pixel. This string contains 12 1-byte characters to display the same

information. At this rate, the P3 PPM will be around four times larger than the binary bitmap.

The P6 version of the PPM file format, is much like a mix of the Windows bitmap and the P3 PPM. The P6's header structure is the same as the P3, with the added constraint that the maximum value of a pixel component cannot exceed 255. The major difference is in the pixel data. The pixel data for the P6 is in a 24-bit binary format like a Windows bitmap, but with the more intuitive layout of a PPM. The P6 was the PPM version used with this research because its less daunting file size and its similarity with the Windows bitmap.

Moving Image File Formats

Of course, the main objective is to process video instead of just still images, so a standard video format had to be chosen. As with the still images, this decision was made when much of the programming was still being done on the Microsoft Windows XP® platform. At the time, the obvious choice seemed to be an uncompressed audio-video interleave format, or AVI. In retrospect, there may have been some other formats that would have been easier to adapt, but none with the flexibility and support of the Windows AVI.

As the name suggests, an AVI file contains audio and video information for every frame. Complete information on the formatting of the Windows AVI was difficult to find, so only basic understanding of the format was achieved. The information obtained was sufficient, though, to create a program that could read and existing AVI, manipulate the pixel information, and save the altered data to a new AVI file. To work properly with this software, AVIs must be carefully built to contain no audio tracks or video compression.

The AVI is written in the Resource Interchange File Format, or RIFF. The RIFF format is commonly used with applications that record, edit, and playback digital media. The RIFF format is a shell that allows for different types of data to be constructed with a common structure. The primary unit of a RIFF is a data chunk. Every bit of information inside a RIFF is contained in a data chunk. Some chunks contain data while other chunks are only used to hold other sub-chunks.

```
RIFF ('AVI '
      LIST ('hdrl'
            'avih'(<Main AVI Header>)
            LIST ('strl'
                  'strh'(<Stream header>)
                  'strf'(<Stream format>)
                  'strd'(<additional header data>)
                  'strn'(<Stream name>)
                  ...
                 )
           )
      LIST ('movi'
            {SubChunk | LIST ('rec '
                              SubChunk1
                              SubChunk2
                                 .
                                 .
                                 .
                             )
            }
           )

      ['idx1'<AVI Index>]
     )
```

Figure 2: Windows AVI Header Data Layout

The basic layout for a Windows AVI is shown in Figure 2. Each chunk has a description consisting of four alphanumeric characters known as the FourCC, or Four Character Code, for that chunk. In addition to the FourCC descriptor, each chunk has a long integer value representing the size of the chunk (size does not include the FourCC descriptor or the chunk size variable).

The RIFF chunk, which is the base form of the file, contains a second FourCC code which indicates which type of RIFF the file is. In this case, the second FourCC

code is ‘AVI ’, indicating the file is an AVI. The main type of container for the RIFF format is the LIST chunk. Chunks labeled to be LISTs are used to group other data chunks together. LIST chunks also have a second FourCC descriptor which serves as a name for the list. The AVI RIFF, must contain two major LIST chunks: a chunk with the header information (‘hdrl’) and a chunk with video data (‘movi’). An optional third chunk that is typically found in AVIs is the file index (‘idx1’) which allows for smoother operation of the AVI. Only the RIFF and LIST chunks can contain sub-chunks [Del02][MDN02].

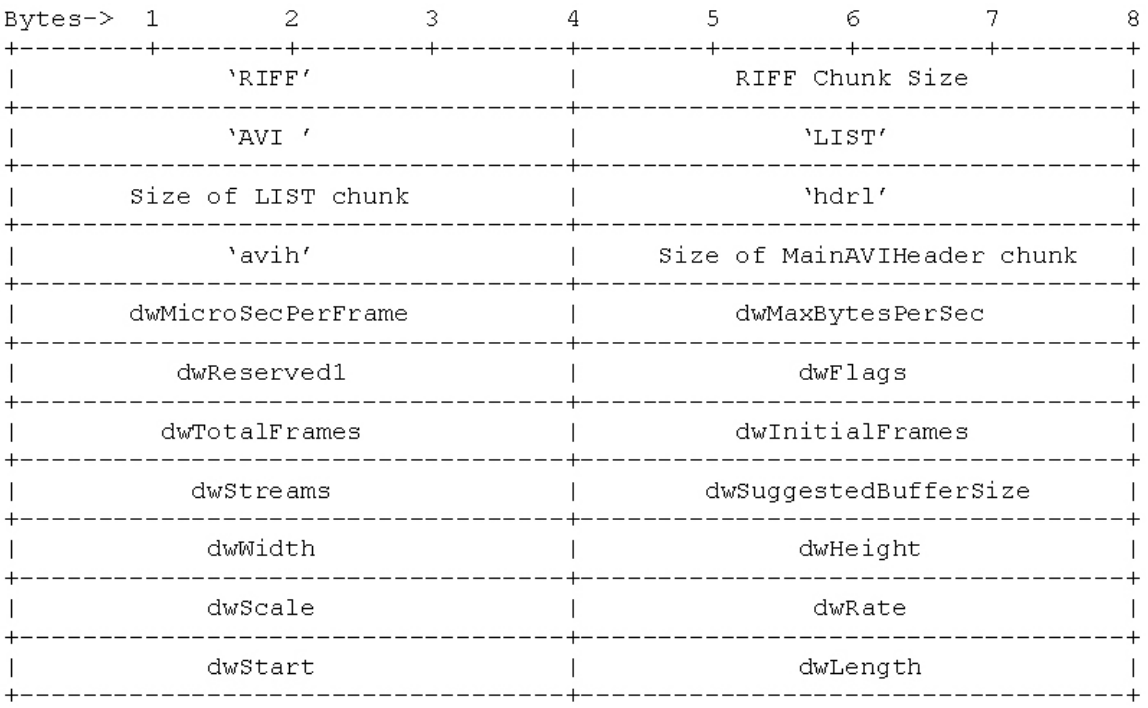

Figure 3: Windows AVI Header Data Layout

There is quite a bit of information that is stored in the ‘hdrl’ LIST chunk, but all that is needed for simple image manipulation is information in the ‘avih’ chunk that contains the main AVI header. The header information in an AVI file is shown from the beginning of the RIFF chunk through the end of the main AVI header in Figure 3. Details on each variable are shown in Table 2.

Table 2: AVI File Header Information

Data Descriptor	Variable Type/Size Offset (bytes) Structure Membership	Description
'RIFF'	FOURCC (4bytes) 0x00 RIFF Chunk	This is the FOURCC base descriptor indicating that the file is in a RIFF format.
RIFF Chunk Size	DWORD (4bytes) 0x04 RIFF Chunk	This value gives the overall size of the base RIFF chunk. This will be close to the entire size of the file. In many cases, the file size is just 8 bytes larger than this value (FileSize – 'RIFF' – RIFF Chunk Size.)
'AVI '	FOURCC (4bytes) 0x08 RIFF Chunk	This is the FOURCC data string that indicates that this RIFF is an AVI.
'LIST'	FOURCC (4bytes) 0x0b LIST 'hdrl' Chunk	This data indicates that this starts a new LIST chunk of data. Only the base RIFF chunk and LIST chunk can have sub-chunks of data
LIST Chunk Size	DWORD (4bytes) 0x10 LIST 'hdrl' Chunk	This is the size of the LIST chunk of data.
'hdrl'	FOURCC (4bytes) 0x14 LIST 'hdrl' Chunk	This FOURCC data string indicates that this LIST chunk contains file and stream headers for the AVI.

Table 2. Continued

Data Descriptor	Variable Type/Size Offset (bytes) Structure Membership	Description
'avih'	FOURCC (4bytes) 0x18 avih Chunk	This code indicates that this is the sub-chunk of the 'hdrl' LIST chunk that contains the main AVI header.
avih Chunk Size	DWORD (4bytes) 0x1b avih Chunk	This value shows the size of the main AVI header.
dwMicroSecPerFrame	DWORD (4bytes) 0x20 MainAVIHeader	This value gives the number of microseconds between each frame and therefore, specifies the overall timing of the file.
dwMaxBytesPerSec	DWORD (4bytes) 0x24 MainAVIHeader	This value is the approximate maximum data rate of the AVI. This information lets the system know many bytes per second it must handle to display an AVI as specified by the other parameters in the file.
dwReserved1	DWORD (4bytes) 0x28 MainAVIHeader	This value is reserved for data padding and should be set to zero.

Table 2. Continued

Data Descriptor	Variable Type/Size Offset (bytes) Structure Membership	Description
dwFlags	DWORD (4bytes) 0x2c MainAVIHeader	This value contains general AVI parameter flags. The individual flags include: *AVIF_HASINDEX* – This flag indicates that the AVI has an 'idx1' index chunk at the end of the file. This index helps to improve display performance. *AVIF_MUSTUSEINDEX* – When this flag is set, the order that AVI data is presented as determined by the index instead of the physical order of the frames in the file. *AVIF_ISINTERLEAVED* – This flag indicates that the AVI is interleaved. *AVIF_WASCAPTUREFILE* – This flag is set when the AVI is a special contiguous file used to capture real-time video. *AVIF_COPYRIGHTED* – This flag indicates that the AVI is copyrighted material.
dwTotalFrames	DWORD (4bytes) 0x30 MainAVIHeader	This value represents the total number of video frames in the file.
dwInitialFrames	DWORD (4bytes) 0x34 MainAVIHeader	This represents the number of initial frames before the start of the stream for the current member in case the audio and video are not synchronized.

Table 2. Continued

Data Descriptor	Variable Type/Size Offset (bytes) Structure Membership	Description
dwStreams	DWORD (4bytes) 0x38 MainAVIHeader	This is the number of data streams in the AVI. An AVI with a video stream and an audio stream will have a value of 2.
dwSuggestedBufferSize	DWORD (4bytes) 0x3c MainAVIHeader	This value is the suggested size of the playback buffer. Generally, this size should be large enough to contain the largest chunk in the file. If this value is set too low, the playback software will have to reallocate memory during playback, which can reduce performance. For an interleaved file, the buffer size should be large enough to read the audio and video data for one frame.
dwWidth	DWORD (4bytes) 0x40 MainAVIHeader	This is the width of the video stream in pixels
dwHeight	DWORD (4bytes) 0x44 MainAVIHeader	This is the height of the video stream in pixels
dwReserved[4]	DWORD [4] (16bytes) 0x44 MainAVIHeader	This is a reserved data block. Elements in this array should be set to zero.

Once the AVI is loaded, all of this information will be available, but only a few of the variables will be needed for processing this video. There are four in particular: the ‘hdrl’ chunk size, dwMicroSecPerFrame, dwTotalFrames, dwWidth, and dwHeight. The first variable, the ‘hdrl’ chunk size, is used to specify where the actual video data startis in the AVI. For AVIs used in this research, the offset stays the same from file to file, but

using the chunk-size data to determine the offset will assure that the correct video data starting position will be found.

The dwMicroSecPerFrame value is used to tell how many frames per second the AVI is. This information is handy to have if any time based functions are to be run on the video data. For instance, to calculate vehicle velocity, the position of the vehicle in two consecutive frames must be known. Then dividing the time difference in the frame by the position distance will give the vehicle's average velocity for that time segment. The other instance in which this information is helpful, is when trying to emulate processing real-time data. When processing real-time data, frames that take too long to process will cause subsequent frames to be missed, making the video seem to jump. This same effect can be achieved with pre-recorded data by inspecting the amount of time that has passed since processing began, and loading the frame that is closest to this time from the video file. This will make the pre-recorded video seem as if it is being processed real-time.

The use of the final three variables is more obvious. The dwWidth and dwHeight variables are needed to determine the resolution of the video stream. The final variable gives the total number of frames in the AVI file, which will let the position system know when to stop processing.

Processing AVI files for feature extraction is done in a couple of ways. Most commonly, the original AVI is opened and read on a by-frame basis into the frame buffer. Any image manipulation or feature extraction is done inside the frame buffer for the frame and then it is discarded. This result may or may not be displayed to the screen, depending on the preference of the user. Data does not need to be written back to a file

for the position system or image processing equations to work properly. This processing method is relatively fast because most of the processing is being done in RAM and the hard drive is only used when a new frame is first read.

Sometimes the processing done on an AVI needs to be saved, typically to create display files. For this type of processing, the only part of the AVI that is being changed are the individual pixels on a frame-by-frame basis. Because of this, the original AVI can be copied verbatim to a new file. Then when image processing is done on the original AVI, the new data can be written overtop of the pixel data in the new AVI. The resolution, frame rate, etc. is going to be the same for both files so this can be done seamlessly without having to know the ins-and-outs of building an AVI.

Processing algorithms for these data types have been created and work well for testing algorithms and the position system as a whole. Real-time video processing has not yet been added as a function to this software, so all testing must be done on simulations or previously recorded test videos. It will be shown throughout this document that this was sufficient to prove this concept works.

CHAPTER 2
IMAGE ENHANCEMENT AND BASIC IMAGE PROCESSING

Image processing is a growing field in which there are extensive methods and techniques to process and enhance visual information. From normal Gaussian distributions, to vector quantization, to neural networks there are many different processing concepts of varying complexity and affectivity. This chapter deals with a few of the simple imaging techniques that can be used readily to enhance or pre-process an image. These simple imaging algorithms typically run very quickly and can sometimes be used in place of the more eloquent ones for processing in undemanding situations.

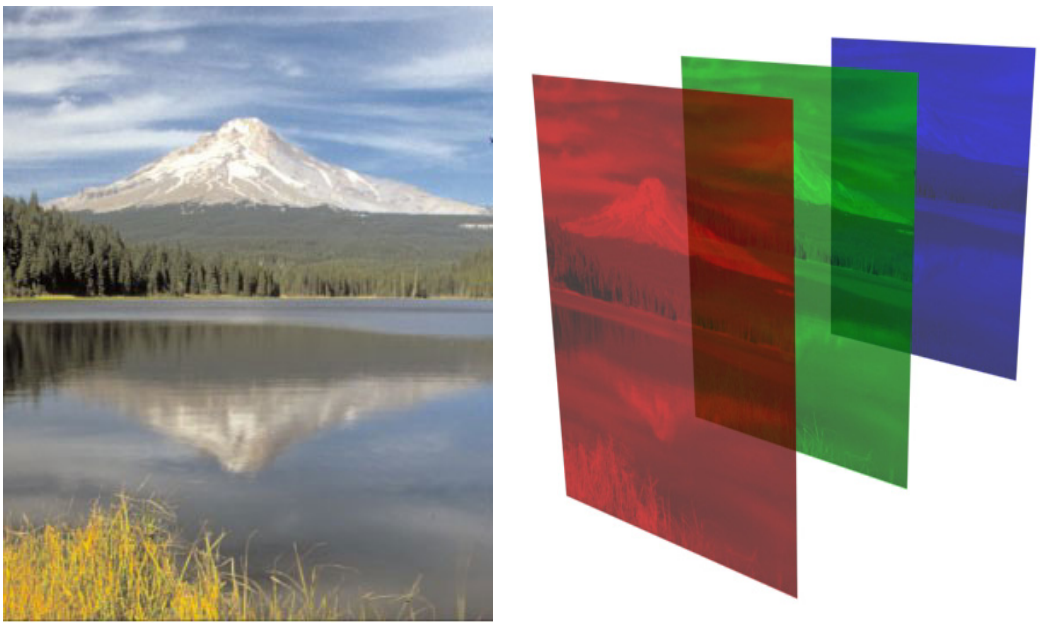

Figure 4: Color Plane Layout. Original Image (Left). Image Split into Red, Green, and Blue Color Planes (Right).

Primitive Imaging Functions

Some basic mathematical definitions and notations must first be covered before discussing individual functions. This will allow for easier explanation of the functions later in the chapter.

All digital images have ($h \times w$) pixels, where h is the height of the image in pixels and the w is the width. Given any 24-bit (true color) image, the corresponding values can be described by the 3-dimensional image matrix $\overline{\overline{I}}$ which has ($h \times w \times 3$) elements. This matrix $\overline{\overline{I}}$ can be described as a 3×1 nested matrix containing the image's red, green, and blue segments portrayed by the individual matrices $\overline{\overline{R}}$, $\overline{\overline{G}}$, and $\overline{\overline{B}}$ (respectively), see Figure 4 . $\overline{\overline{R}}$, $\overline{\overline{G}}$, and $\overline{\overline{B}}$ each have w columns and h rows.

$$\overline{\overline{I}} = \begin{bmatrix} \overline{\overline{R}} \\ \overline{\overline{G}} \\ \overline{\overline{B}} \end{bmatrix} \tag{1}$$

For the following algorithms to work properly the data must be formatted in the fashion shown. Not all image formats will have data packaged in this way. This must be dealt with by the function reading the image or video file. The imaging functions have no way of dealing with data that is structured improperly. The format of $\overline{\overline{R}}$, $\overline{\overline{G}}$, and $\overline{\overline{B}}$ in a properly formatted data array should be as follows:

$$\overline{\overline{R}} = \begin{bmatrix} R_{0,0} & R_{0,1} & \cdots & R_{0,w-1} \\ R_{1,0} & R_{1,1} & \cdots & R_{1,w-1} \\ \vdots & \vdots & \ddots & \vdots \\ R_{h-1,0} & R_{h-1,1} & \cdots & R_{h-1,w-1} \end{bmatrix} \tag{2}$$

$$\overline{\overline{G}} = \begin{bmatrix} G_{0,0} & G_{0,1} & \cdots & G_{0,w-1} \\ G_{1,0} & G_{1,1} & \cdots & G_{1,w-1} \\ \vdots & \vdots & \ddots & \vdots \\ G_{h-1,0} & G_{h-1,1} & \cdots & G_{h-1,w-1} \end{bmatrix} \quad (3)$$

$$\overline{\overline{B}} = \begin{bmatrix} B_{0,0} & B_{0,1} & \cdots & B_{0,w-1} \\ B_{1,0} & B_{1,1} & \cdots & B_{1,w-1} \\ \vdots & \vdots & \ddots & \vdots \\ B_{h-1,0} & B_{h-1,1} & \cdots & B_{h-1,w-1} \end{bmatrix} \quad (4)$$

Each element of the preceding matrices is a scalar, unsigned, 8-bit integer. Therefore, each value is constrained to a range of zero to 255. A component value of zero means that that particular pixel has a minimum intensity for that component color, whereas, a value of 255 means that the pixel has full component intensity.

For subsequent calculations; $R_{i,j}$, $G_{i,j}$, and $B_{i,j}$ correspond to the red, green, and blue color intensities (respectively) for the pixel at row i and column j of the image. When a function's user defined variables are present in the equation, they will be referenced as a function abbreviation with the subscript showing the name of the variable. For example, CB_{Red} would be the value of the variable 'Red' in the *ColorBias* function. The names of the variables should be fairly intuitive, but the variable name and description can be found in the 'C Code Description' of that function at the end of each section.

Color Biasing

Color biasing enhances an image by changing the effect of the primary colors (red, green, and blue) on the original image. Biasing the red, green, or blue color planes changes the dominance of that color in every pixel in the image. This allows the user to modify color distributions in the image or change the overall brightness of an image.

Usage

To modify the brightness of an image, color bias must be run on all the color planes together, see Figure 5. Bright images can be darkened with color bias and dark images can be lightened. The distribution of the colors with respect to each other changes very little when all color planes are modified so the effect is mainly aesthetic. While brightening an image may help a human viewer discern some object features, it is unlikely that other image processing algorithms would benefit from these adjustments.

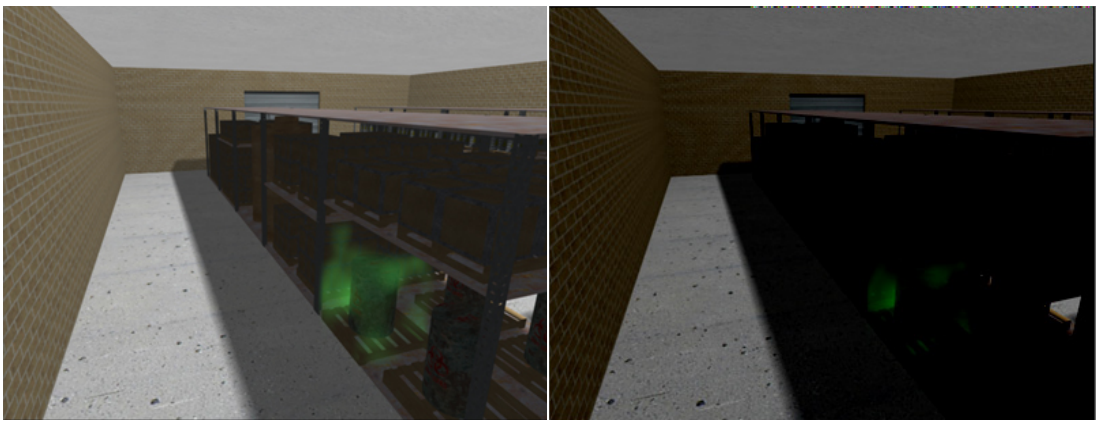

Figure 5: Original Image (Left). Image with Brightness Lowered with Color Bias (Right).[1]

Color bias used separately on color planes can be used to remove the effects of undesirable lighting from an image. Lights that are any color other than white will cast a colored hue on objects in the scene. Color biasing can be done to remove these ambient lighting effects without destroying the individual features in the original image. In Figure 6, the original image is saturated with sunlight that casts a yellow hue on the scene. The right hand image has been color biased to remove the yellow from the image. The change to the original pixel ratios is minimal. This allows the feature extraction algorithms run on the image afterward to still yield accurate results. Color biasing is used

[1] Right image is color biased with RGB shift values of (-50,-50,-50)

primarily for image enhancement not for feature extraction, but it is a useful tool nonetheless.

Figure 6: Original Image with Yellow Hue from the Sun (Left). Color Biased Image with Yellow Hue Removed (Right).[2]

Mathematics

For every pixel in the image:

$$\begin{aligned} R_{i,j} &\leftarrow R_{i,j} + CB_{red} \\ G_{i,j} &\leftarrow G_{i,j} + CB_{grn} \\ B_{i,j} &\leftarrow B_{i,j} + CB_{blu} \end{aligned} \tag{5}$$

Where CB_{red}, CB_{grn}, and CB_{blu} correspond to the red shift, green shift, and blue shift of the *ColorBias* function respectively. To ensure that all values stay within the range, any resulting values of $R_{i,j}$, $G_{i,j}$, and $B_{i,j}$ that exceed 255 are set to 255 and any values less than zero are set to zero.

C code reference

```
ColorBias
(Last Modified on 01/08/03)

Description:
Adjusts the presence of the individual color planes of the
  image.  Works similar to a brightness control for the red,
```

[2] Right image is color biased with RGB shift values of (-30,-30,0).

```
  green and blue color planes.  A pixel with no red that is
  color biased with a red value of 255 will give the pixel
  the maximum amount of red possible.  A value of -255 will
  remove all the red from a pixel with that starts with the
  maximum red possible.  A value of 0 will leave the red
  color plane unaltered.

Usage:
int ColorBias(unsigned int Width, unsigned int Height,
  unsigned char *PixelArray, int Red, int Grn, int Blu)

   -  Width       - Width of the frame in pixels
   -  Height      - Height of the frame in pixels
   -  *PixelArray - Pointer to the pixel data array
   -  Red         - Red color plane shift
   -  Grn         - Green color plane shift
   -  Blu         - Blue color plane shift

Returns:  0  for successful
         -1  on error

Requirements:
VisFcns.h
```

Color Distinguishing

Color distinguishing is the process of searching an image for a color range and filtering out any colors that do not match the selection. This is done by looking for a target color range that is identified by a red, green, and blue value and filtering out (set to 0) any pixels that do not fall within a user-set threshold of that search value.

Usage

Color distinguishing is a simple and fast algorithm that can be used for feature extraction in some simpler scenarios. Color distinguishing allows the user to remove extraneous data from the image. In Figure 7, white lines have been detected by using color distinguishing. The grass and the orange stripes on the barrel have been removed leaving only the white portions of the image. Color distinguishing, when used in

conjunction with some other primitive image processing algorithms, can take the place of some of the more complex statistical and training based models.

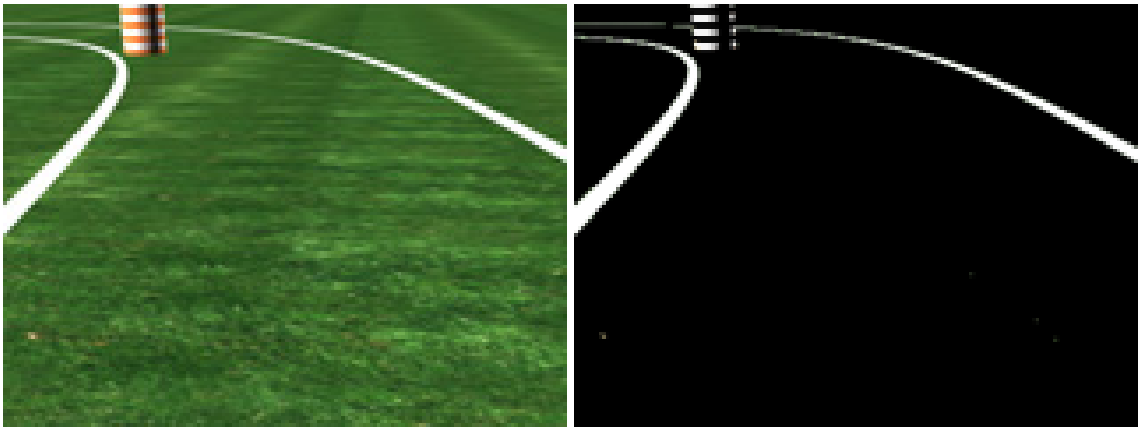

Figure 7: Original Image (Left). Color Distinguished Image Where White has been used as the Target Color (Right).[3]

Unfortunately, to be effective, color distinguishing must be done only when lighting conditions are fairly constant and the color of the feature being distinguished is unique in the image. If lighting conditions change, the colors of the pixels being searched for will change as well and could likely fool the algorithm. If the feature color is not unique, there may be false positives that are found along with the feature. Therefore, in real world applications, color distinguishing should generally not be used.

Mathematics

For every pixel in the image:

$$R_{i,j} \leftarrow \begin{cases} null & if \quad CD_{red} + CD_{threshold} < R_{i,j} \\ null & if \quad CD_{red} - CD_{threshold} > R_{i,j} \\ R_{i,j} & otherwise \end{cases} \tag{6}$$

$$G_{i,j} \leftarrow \begin{cases} null & if \quad CD_{grn} + CD_{threshold} < G_{i,j} \\ null & if \quad CD_{grn} - CD_{threshold} > G_{i,j} \\ G_{i,j} & otherwise \end{cases} \tag{7}$$

[3] Right image is color distinguished with RGB target values of (255,255,255) and a threshold value of 50.

$$B_{i,j} \leftarrow \begin{cases} null & if \quad CD_{blu} + CD_{threshold} < B_{i,j} \\ null & if \quad CD_{blu} - CD_{threshold} > B_{i,j} \\ B_{i,j} & otherwise \end{cases} \tag{8}$$

If $R_{i,j}$, $G_{i,j}$, or $B_{i,j}$ return a null value then $R_{i,j} = G_{i,j} = B_{i,j} = 0$. If no null's are returned then the original pixel is kept.

C code reference

```
ColorDistinguish
(Last Modified on 07/30/02)

Description:
Singles out a color range and eliminates all but that color
  range from the image.

Usage:
int ColorDistinguish(unsigned int Width, unsigned int
  Height, unsigned char *PixelArray, unsigned int Red,
  unsigned int Grn, unsigned int Blu, unsigned int
  Threshold)

   - Width       - Width of the frame in pixels
   - Height      - Height of the frame in pixels
   - *PixelArray - Pointer to the pixel data array
   - Red         - Red component of target distinguish
     color
   - Grn         - Green component of target distinguish
     color
   - Blu         - Blue component of target distinguish
     color
   - Threshold   - +/- range for the selected color

Returns: 0  for successful
        -1  on error

Requirements:
VisFcns.h
```

Color Removal

Color removal is the compliment to color distinguishing. A target color and threshold is entered by the user and the selected color range is removed from the image.

Any pixels in this color range will be removed from image (set to 0) while all others remain intact.

Usage

Color removal is done when certain known aspects of the scene must be removed from the image. Removing unnecessary colors from an image can speed up the use of more complicated algorithms later on. For instance, removing colors that are not needed can speed up the segmentation of an image, which can be a lengthy process when there are many colors in an image. Figure 8 shows an example of color removal where white road lines have been removed from the scene.

Figure 8: Original Image (Left). Color Removed Image Where White has been used as the Target Color (Right).[4]

Mathematics

Just like in color distinguish; there are three equations that determine the outcome of the each pixel in the image:

$$R_{i,j} \leftarrow \begin{cases} R_{i,j} & if \quad CD_{red} + CD_{threshold} < R_{i,j} \\ R_{i,j} & if \quad CD_{red} - CD_{threshold} > R_{i,j} \\ null & otherwise \end{cases} \tag{9}$$

[4] Right image is color removed with RGB target values of (255,255,255) and a threshold value of 50.

$$G_{i,j} \leftarrow \begin{cases} G_{i,j} & if \quad CD_{grn} + CD_{threshold} < G_{i,j} \\ G_{i,j} & if \quad CD_{grn} - CD_{threshold} > G_{i,j} \\ null & otherwise \end{cases} \tag{10}$$

$$B_{i,j} \leftarrow \begin{cases} B_{i,j} & if \quad CD_{blu} + CD_{threshold} < B_{i,j} \\ B_{i,j} & if \quad CD_{blu} - CD_{threshold} > B_{i,j} \\ null & otherwise \end{cases} \tag{11}$$

Like in color distinguish; if $R_{i,j}$, $G_{i,j}$, or $B_{i,j}$ return a null value then $R_{i,j} = G_{i,j} = B_{i,j} = 0$. If no null's are returned then the original pixel is kept.

C code reference

```
ColorRemove
(Last Modified on 07/30/02)

Description:
Removes a color range from an image.

Usage:
int ColorRemove(unsigned int Width, unsigned int Height,
  unsigned char *PixelArray, unsigned int Red, unsigned int
  Grn, unsigned int Blu, unsigned int Threshold)

   - Width       - Width of the frame in pixels
   - Height      - Height of the frame in pixels
   - *PixelArray - Pointer to the pixel data array
   - Red         - Red component
   - Grn         - Green component
   - Blu         - Blue component
   - Threshold   - +/- range from the central value

Returns:  0  for successful
         -1  on error

Requirements:
VisFcns.h
```

Thresholding

Thresholding is a technique that involves dropping pixels that have red, green, and blue intensities less than a user set value. Threshold examines the red, green, and blue color planes separately. This allows for part of a pixel to be dropped without affecting the rest of the pixel. Therefore if a pixel does not have enough red to surpass the user set threshold for red, the red portion of the pixel is set to zero, but the green and blue remain intact. The net effect of this operation is a modification of the original color instead of the removal of the pixel. This allows for some different effects to be obtained with threshold post-processing.

The thresholding algorithm comes in three different varieties. The first is simple thresholding with no post-processing. This allows the user to drop darker color values that may not be needed for feature detection and runs fairly quickly. The second, color extrapolation, stretches the colors that are not initially dropped over the full available color range. This post-processing darkens the image, but shows more contrast between individual colors. The final type of thresholding, color maximization, takes pixels that remain after thresholding and sets any non-zero values to full intensity. Examples of all three types of thresholding can be seen in Figure 9.

Usage

Each of the types of thresholding has different potential usages. The basic thresholding is used when darker, less vibrant pixels need to be removed to aid in feature extraction. Removal of these pixels can speed up and simplify calculations for subsequent enhancement algorithms.

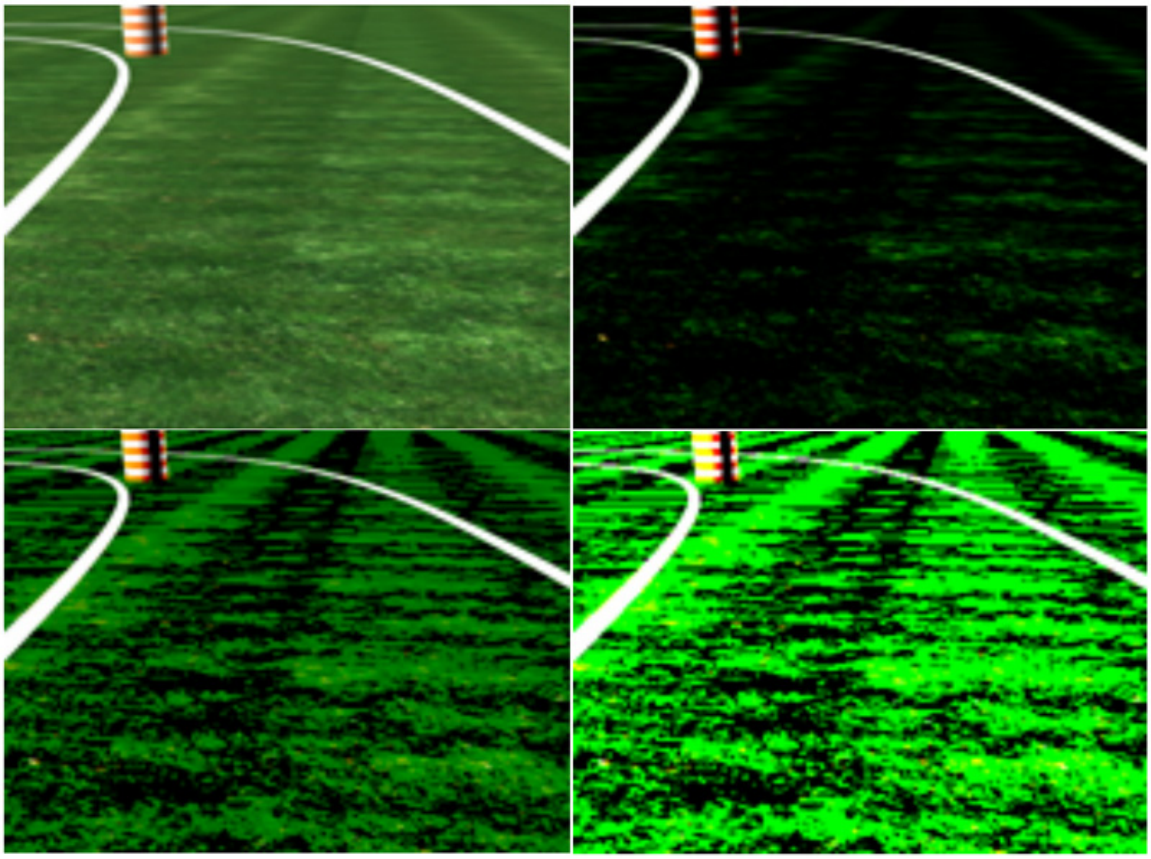

Figure 9: Original Image (Top-Left). Image Processed with No Post-Thresholding using *Threshold_Drp* algorithm (Top-Right). Image Processed with Color Extrapolation using *Threshold_Str* algorithm (Bottom-Left). Image Processed with Color Maximization using *Threshold_Max* algorithm (Bottom-Right).[5]

In addition to removing dark background pixels, the next type of thresholding stretches the colors and increases minor differences between remaining pixels. This can be used to enhance patterns that are similar in color. For instance, wood grain texture that is barely noticeable in an original image will stand out once the colors have been stretched out, see Figure 10.

The final type, maximization thresholding, simplifies individual colors and primitively segments and image. This type of thresholding groups all remaining colors

[5] Top-right, bottom-left, and bottom-right images were thresholded with RGB minimum values of (50,50,50).

into one of seven different colors. For example, if red is being scanned for in the image, the maximization will group all different shades of red into a single color for easy detection. The result is an image consisting of only primary (red, green, and blue) and secondary (white, black, yellow, cyan, and magenta) colors. Also, the maximization variety of thresholding can be used to convert an image to a simple 3-bit color format, see Figure 11.

Figure 10: Original Image (Left). Image Thresholded to Enhance Wood Grain (Right).[6]

Mathematics

The basic thresholding calculations are done on the first two varieties. For every pixel in the image:

$$R_{i,j} \leftarrow \begin{cases} 0 & if \quad TH_{red} > R_{i,j} \\ R_{i,j} & otherwise \end{cases} \tag{12}$$

$$G_{i,j} \leftarrow \begin{cases} 0 & if \quad TH_{grn} > G_{i,j} \\ G_{i,j} & otherwise \end{cases} \tag{13}$$

[6] Image on right thresholded using *Threshold_Str* algorithm with red, green, blue values of (120,120,120).

$$B_{i,j} \leftarrow \begin{cases} 0 & if \quad TH_{blu} > B_{i,j} \\ B_{i,j} & otherwise \end{cases} \tag{14}$$

Figure 11: Original Image (Left). Thresholded Image Converted to 3-bit Color (Right).

These first calculations drop the values of pixels that fall below the threshold set by the user. For the standard thresholding, *(Threshold_Drp)* algorithm, these are the only conditions checked. For the thresholding with color extrapolation, *(Threshold_Str)*, the following extrapolation equations are run on the pixel components with non-zero values:

$$R_{i,j} \leftarrow \frac{255 \cdot (R_{i,j} - TH_{red})}{255 - TH_{red}} \tag{15}$$

$$G_{i,j} \leftarrow \frac{255 \cdot (G_{i,j} - TH_{grn})}{255 - TH_{grn}} \tag{16}$$

$$B_{i,j} \leftarrow \frac{255 \cdot (B_{i,j} - TH_{blu})}{255 - TH_{blu}} \tag{17}$$

The pixels with color values equal to the threshold will become zero and the rest of the values will be spread proportionally up to 255. After the extrapolation, some values may not be integers, so the new values must be truncated:

$$R_{i,j} \leftarrow \lfloor R_{i,j} \rfloor \tag{18}$$

$$G_{i,j} \leftarrow \lfloor G_{i,j} \rfloor \quad (19)$$

$$B_{i,j} \leftarrow \lfloor B_{i,j} \rfloor \quad (20)$$

Where $\lfloor R_{i,j} \rfloor$ represents the *floor* value of $R_{i,j}$ or the closest integer with a value less than or equal to $R_{i,j}$. The final type of thresholding, color maximization *(Threshold_Max)*, uses modified threshold equations:

$$R_{i,j} \leftarrow \begin{cases} 0 & if \quad TH_{red} > R_{i,j} \\ 255 & otherwise \end{cases} \quad (21)$$

$$G_{i,j} \leftarrow \begin{cases} 0 & if \quad TH_{grn} > G_{i,j} \\ 255 & otherwise \end{cases} \quad (22)$$

$$B_{i,j} \leftarrow \begin{cases} 0 & if \quad TH_{blu} > B_{i,j} \\ 255 & otherwise \end{cases} \quad (23)$$

The result is an image of color components that are either on or off.

C code reference

```
Threshold_Drp
Threshold_Str
Threshold_Max
(Last Modified on 07/30/02)

Description:
All three thresholds drop pixel components that are less
  then the values specified by Red, Grn, and Blu.

   - Threshold_Drp (Drop) thresholds only.
   - Threshold_Str (Stretch) scales the values that remain
     across the entire range (0-255).
   - Threshold_Max (Max) takes the remaining values and
     sets them all to 255.

Usage:
int Threshold_Drp(unsigned int Width, unsigned int Height,
  unsigned char * PixelArray, unsigned int Red, unsigned int
  Grn, unsigned int Blu)
```

```
   -  Width       - Width of the frame in pixels
   -  Height      - Height of the frame in pixels
   -  *PixelArray - Pointer to the pixel data array
   -  Red         - Minimum value for the red bit-plane
   -  Grn         - Minimum value for the green bit-plane
   -  Blu         - Minimum value of the blue bit-plane

Returns: 0  for successful
        -1  on error
Requirements:
VisFcns.h
```

Edge Detection

The edge detection developed in this software library searches for image edges, not necessarily object edges. Edges are determined by color disparity between adjacent pixels. Each color plane is searched for edges independently so that any discrepancy between the pixels is accounted for. The outcome of the edge detection is placed back into the original pixel location. The resulting color depends on the original colors compared, and thus, edge colors correspond to original color pairs. This property allows the user to search for a border between two colors by searching for the corresponding edge color.

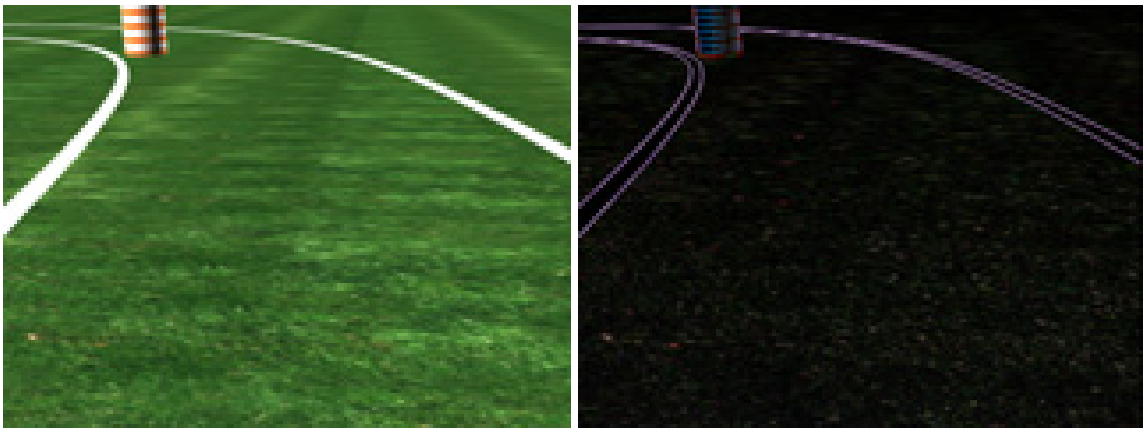

Figure 12: Original Image (Left). Edge Detected Image with No Pre-Processing (Right).[7]

[7] Right image was edge detected using the 'Dual' method.

Usage

Edge detection, while very simple, is perhaps the most powerful of the primitive image processing tools in this software library. Edge detection can be used to find the outline of an object in the scene, as well as, features on an object. Figure 12 shows an image that has been edge detected. In this example the only algorithm run on the image is edge detection. Typically some pre-processing would be done before the edge detection to limit the amount of edge noise found as a result of pixel aliasing.

The EdgeDetect algorithm has three very similar variations available for use. The simple horizontal edge detection *(EdgeDetect_Horz)* takes the discrepancy between each of the color components of the current pixel and the components of its neighbor to the right. This is done for the red, green, and blue color components. This is done sequentially for the entire image. This algorithm runs fast, but horizontal lines do not get detected. This is because the color difference is running perpendicular to the direction that is being inspected.

For inspecting image features such as road lines, which should rarely be perpendicular to the camera, this type of edge detection might be feasible. On the other hand, if the image being inspected is of a box on a conveyor belt; the probability of missing a box's edge is relatively high. Therefore, this type of edge detection should only be used when missing horizontal edges is not a problem.

The next type of edge detection *(EdgeDetect_Diag)* inspects discrepancies between pixels diagonal to each other. Just like in horizontal edge detection, edges that are perpendicular to the detection direction are missed. This type of detection when used on detecting road lines yielded very high errors during curves. Once again, use of this algorithm is only warranted when detection of diagonal lines is unnecessary.

The solution of finding all edges in an image was a combination of two different edge detections. This dual type of edge detection *(EdgeDetect_Dual)* checks pixel color differences between the current pixel and the one to its right. Then it takes the difference of the current pixel and the one underneath. These two values are then averaged to get the total edge value for that pixel. This dual detection is the most reliable of the edge detection algorithms, but it is also the slowest. Dual detection takes more than twice as long to run per image, so the added reliability does not come without a price.

Mathematics

In all three cases of edge detection, the image that is created is smaller than the original. There can only be *n* - 1 sequential differences take from *n* values. Technically these differences represent the space between the two compared pixels. Since it is not possible to show this in the image, the new data is placed back in the original pixel. This is done successively to the right and down the array, with the data placed in the top, left of the pixels being inspected. In this fashion, previously taken differences don't get recalculated. The result is an edge detected image that is shifted up and to the left of ½ pixel. For the horizontal edge detection the following equations are used:

$$R_{i,j} \leftarrow abs(R_{i,j} - R_{i,j+1}) \quad j = 0 \ldots w-2 \tag{24}$$

$$G_{i,j} \leftarrow abs(G_{i,j} - G_{i,j+1}) \quad j = 0 \ldots w-2 \tag{25}$$

$$B_{i,j} \leftarrow abs(B_{i,j} - B_{i,j+1}) \quad j = 0 \ldots w-2 \tag{26}$$

The diagonal edge detection uses a similar set of equations:

$$R_{i,j} \leftarrow abs(R_{i,j} - R_{i+1,j+1}) \quad i = 0 \ldots h-2 \quad j = 0 \ldots w-2 \tag{27}$$

$$G_{i,j} \leftarrow abs(G_{i,j} - G_{i+1,j+1}) \quad i = 0 \ldots h-2 \quad j = 0 \ldots w-2 \tag{28}$$

$$B_{i,j} \leftarrow abs(B_{i,j} - B_{i+1,j+1}) \quad i = 0 \ldots h-2 \quad j = 0 \ldots w-2 \tag{29}$$

The dual edge detection is slightly more complex:

$$R_{i,j} \leftarrow \frac{abs(R_{i,j} - R_{i,j+1}) + abs(R_{i,j} - R_{i+1,j})}{2} \quad i = 0 \ldots h-2 \quad j = 0 \ldots w-2 \tag{30}$$

$$G_{i,j} \leftarrow \frac{abs(G_{i,j} - G_{i,j+1}) + abs(G_{i,j} - G_{i+1,j})}{2} \quad i = 0 \ldots h-2 \quad j = 0 \ldots w-2 \tag{31}$$

$$B_{i,j} \leftarrow \frac{abs(B_{i,j} - B_{i,j+1}) + abs(B_{i,j} - B_{i+1,j})}{2} \quad i = 0 \ldots h-2 \quad j = 0 \ldots w-2 \tag{32}$$

C code reference

```
EdgeDetect_Horz
EdgeDetect_Diag
EdgeDetect_Dual
(Last Modified on 07/30/02)

Description:
Finds object edges by using pixel differentials.

  - EdgeDetect_Horz (Horizontal) determines differentials
    between pixels in the same row.  Scan is fast, but
    does not detect horizontal lines.
  - EdgeDetect_Diag (Diagonal) determines pixel
    differentials between pixels that are diagonal from
    each other.  This scan is also fast but does not
    recognize 45 degree lines.
  - EdgeDetect_Dual (Dual) scans differentials
    horizontally and then vertically and averages the two.
    This scan finds all edges, but takes more than

Usage:
int EdgeDetect_Dual(unsigned int Width, unsigned int
  Height, unsigned char * PixelArray)

  - Width       - Width of the current frame in pixels
  - Height      - Height of the current frame in pixels
  - *PixelArray - Pointer to the pixel data array

Returns: 0  for successful
        -1  on error
```

```
Requirements:
VisFcns.h
```

Image Smoothing

Image smoothing is a process that takes a sharp image and softens it by averaging the color information over consecutive pixels. Each pixel's color information is compared with its neighbor and weighted average of the colors is made. The result is a pixel that contains a fraction of the color information of its surrounding pixels. The stronger the contrast between pixels, the greater effect the smoothing has on the image. The final result is a slightly blurred, but much smoother image.

Usage

Smoothing an image can greatly reduce the random variations in an image that occur. For instance, in Figure 13 the image is smoothed to reduce the effect of the random color variation created by the grass.[8] Typically, there would be dissimilarity between many of the neighboring pixels. Smoothing the image slightly blurs these differences and makes the related areas look more cohesive.

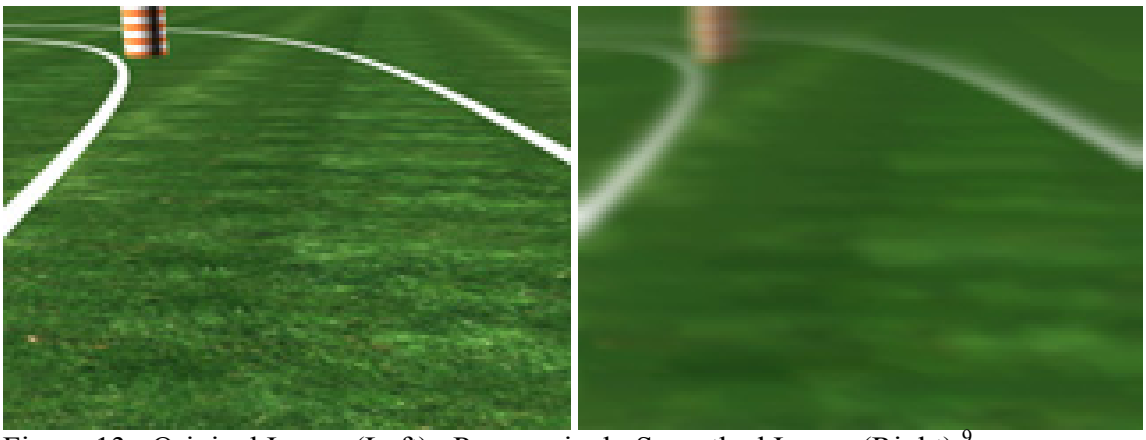

Figure 13: Original Image (Left). Progressively Smoothed Image (Right).[9]

[8] This image has been antialiased numerous times to create a more visible effect. Typical usage would not create this kind of image distortion.

[9] Right image was smoothed with a pixel size of 5.

ProgressiveSmooth can be used to handle images that have been aliased by the capture device. This works especially well if only the less sophisticated functions are being used. For instance, if a raw image is being edge detected, there will almost certainly be edge noise found. Smoothing the image first can greatly reduce the noise while leaving the real edges virtually unaffected.

Mathematics

Progressive smooth is a simple averaging of consecutive pixels. Starting from the top left part of the image and working down and to the right using the equation:

$$R_{i,j} \leftarrow \frac{\sum_{m=0}^{N}\sum_{n=0}^{N} R_{i+m,j+n}}{N^2} \tag{33}$$

$$G_{i,j} \leftarrow \frac{\sum_{m=0}^{N}\sum_{n=0}^{N} G_{i+m,j+n}}{N^2} \tag{34}$$

$$B_{i,j} \leftarrow \frac{\sum_{m=0}^{N}\sum_{n=0}^{N} B_{i+m,j+n}}{N^2} \tag{35}$$

ProgressiveSmooth's processing amount is user set by specifying the number of row and columns to average at one time. In most instances, a smoothing amount of two is sufficient. This will create new pixels by averaging the pixels in a 2×2 formation and placing the result in the top, left element. At this setting the algorithm runs quickly and smoothing is minimal. The user can set any integer as the amount as long as it is less than the height and width of the current frame. As the smoothing amount increases, the time needed for calculations increases exponentially. This makes using a smooth amount of more than three or four impracticable.

C code reference

```
ProgressiveSmooth
(Last Modified on 08/07/02)

Description:
Algorithm uses a running average to soften image and reduce
  noise.

Usage:
int ProgressiveSmooth(unsigned int Width, unsigned int
  Height, unsigned char *PixelArray, unsigned int Amount)

   - Width       - Width of the current frame in pixels
   - Height      - Height of the current frame in pixels
   - *PixelArray - Pointer to the pixel data array
   - Amount      - Number of consecutive pixels to average

Returns 0 for successful

Requirements:
VisFcns.h
```

Primitive Image Processing Example

For some simple cases, primitive processing is enough to extract desired information from an image. The images must be fairly consistent and uncomplicated for the non-statistical models to work. An example of simple image processing will be shown in this section.

Trilateration Background

This example was taken from a peer's research in which some of these algorithms are being used [Yel03]. The process, described as *trilateration*, involves calculating global position by relating distance to a series of landmarks in known locations. Using visual data from the image, the position and orientation of the landmark can be found. When this information is found for three landmarks simultaneously, the camera's position can be extracted. Since the cameras would be mounted on a vehicle, the vehicle's

position would be known as well. This method would be used in place of a global positioning system (GPS) or an inertial measurement unit (IMU).

Figure 14: Original Picture of Trilateration Landmark

The landmark that is suggested for use in this experiment is a sphere with a radius of at least a meter, see Figure 14. This gives a large landmark that looks similar from all positions of the camera. Trilateration involves determining the orientation of the camera by examining the colors present on the landmark at the current angle. The proportions of the colors present give the rotational angle of the camera about the vertical axis of the landmark.

The second rotational value is found by examining the equator of the sphere. When the camera is exactly level with the landmark the equator should appear as a line. When the camera is higher than the sphere the equator should appear as an arc facing upward. The opposite is true when the camera is lower than the landmark. Finding the apparent radius and direction of this arc gives the second rotational value. The distance is the final value needed to get complete 3-dimensional coordinates of the camera. This distance is found by determining the diameter of the sphere in the image. This can be directly correlated to the camera distance from the landmark. Clearly, to determine all of the information required, a good deal of image processing must be done. This example is going to show one part of the processing that must be done to retrieve all of this data.

Image Processing

The image contains many different environmental and hardware conditions that prohibit immediate measurements of the sphere. The first that will be dealt with is the blue hue that can be seen in the image (see Figure 14). This hue was caused by the old model CCD camera that was being used. While this hue does not greatly affect the image, it was removed to exemplify using color biasing for hue correction, see Figure 15.

The next step is to try to isolate the sphere from the rest of the background image. There are several different items here that must be removed; the trees, the sky, the building on the right, etc. Since all of the items that need to be removed are so different, it will be easier to instead just find the items that we want. For this particular image we should only need the red. To do this *ColorDistinguish* is run on the image, see Figure 15.

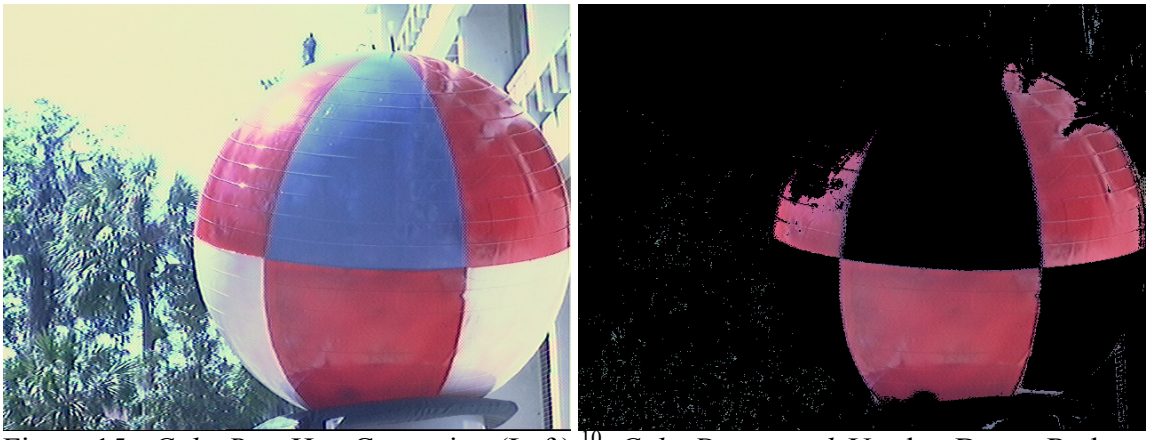

Figure 15: *ColorBias* Hue Correction (Left).[10] *ColorDistinguish* Used to Detect Red (Right).[11]

Here almost the entire image has been removed. All that remains is the part of the ball that appeared red in the image. Unfortunately, the parts of the ball that were reflecting other things were removed as well. On the left side of the image some image

[10] Image was modified by ColorBias with red, green, and blue values of (0,0,-50)

[11] Image was modified by ColorDistinguish with red, green, blue, and threshold values of (255,0,0,150)

noise has remained from the trees that originally occupied this part of the frame. These specks that remain are close enough to the red of the ball that they were unintentionally picked up by the algorithm.

The next issue is to remove the noise left by *ColorDistinguish*. Running *ProgressiveSmooth* reduced the noise by smearing the lit pixels with the black surrounding regions. This is shown in Figure 16, but the effect is hard to see on the scaled down images in the document.

The final step is to intensify the parts of the image that are desired and to completely remove the noise. Since the noise was softened by *ProgressiveSmooth*, it is now easy to drop with thresholding. The *Threshold_Max* algorithm drops all the remaining noise, but also sets the red parts of the ball to maximum intensity. The image data at this point is at an all-or-nothing state which makes the use of any specific detection algorithms run smooth and quick. Figure 16 shows the final result.

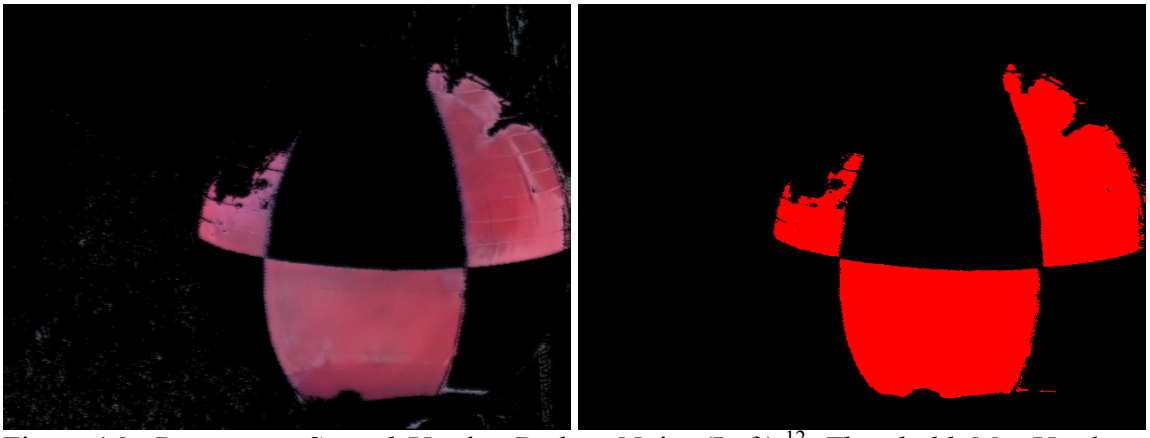

Figure 16: *ProgressiveSmooth* Used to Reduce Noise (Left).[12] *Threshold_Max* Used to Intensify Red (Right).[13]

[12] Image was modified by ProgressiveSmooth with a smoothing amount of 2.

[13] Image was modified by Threshold_Max with red, green, and blue values of (150,255,255)

The image is now filtered down to a point where data is either true or false. Now subsequent algorithms only need to check the red color plane for non-zero values, because all unnecessary data has be set to zero.

CHAPTER 3
STATISTICAL IMAGE PROCESSING

The image processing and enhancement techniques discussed in the last chapter can be useful for some minor adjustments on an image, but for reliable feature extraction, more in-depth approaches must be used. Searching an image for a color, or range of colors, is rarely an effective type of processing. Data must be looked at for many scenarios and a statistical, training-based, or more complicated model must be used for the processing to be effective.

This chapter deals with a few of the statistical image classifiers along with their strong and weak points. Multiple approaches are needed to account for the many different aspects of this research. Some models work well in one scenario and horribly in another, while others are moderately good for all scenarios. For the best results, combinations of models can be used at times to increase reliability and improve performance.

Image Color Distributions

Information in an image can come from a variety of sources. The overall brightness of an image might contain some useful information, while the gradient of color changes between pixels may contain other important data. Even the change in color of a pixel, with relation to time in a video stream, can yield facts about a scene. For this research, though, the color distributions in the image hold the information needed for feature extraction.

Color Histograms

The distribution of intensities within a color channel can be just as useful as the relationships between colors. To see how each color is distributed throughout an image, a histogram of each color channel can be made. Figure 17 shows an example image of the ocean, as well as, its red, green, and blue color channels.

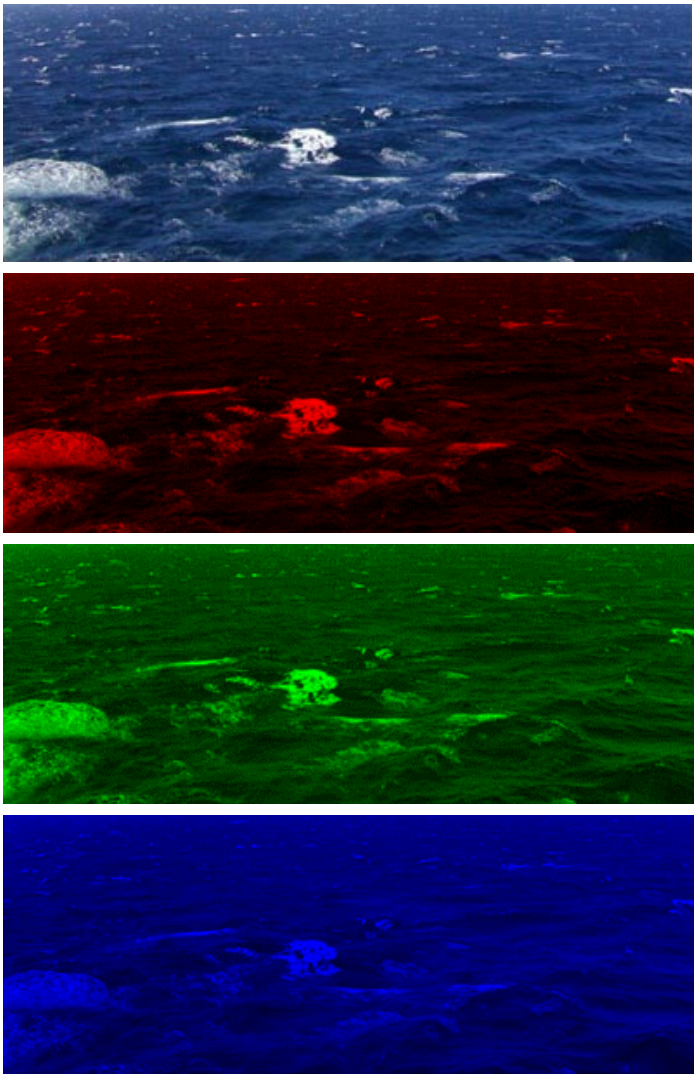

Figure 17: Sample Image of the Ocean: Original Image, Red Color Channel, Green Color Channel, and Blue Color Channels (from Top to Bottom).

Ocean water, which is comprised of mostly blue with a hint of green, has the majority of its color information in those two channels. This is apparent through the brightness of each of the images in relation to each other.[1] The blue is very bright, with very few darker regions. The green shows slightly more contrast. The red, which is the least important color in the image, is very dark in comparison to the other two.

[1] Printed copies of this document may not show image colors adequately.

Basic information can be derived from the color channel images, but it would be beneficial to quantify these values. A set of histograms can be created to show the distribution of the intensities for each color channel. This information can be very useful in determining the type of statistical model to use when searching for certain features.

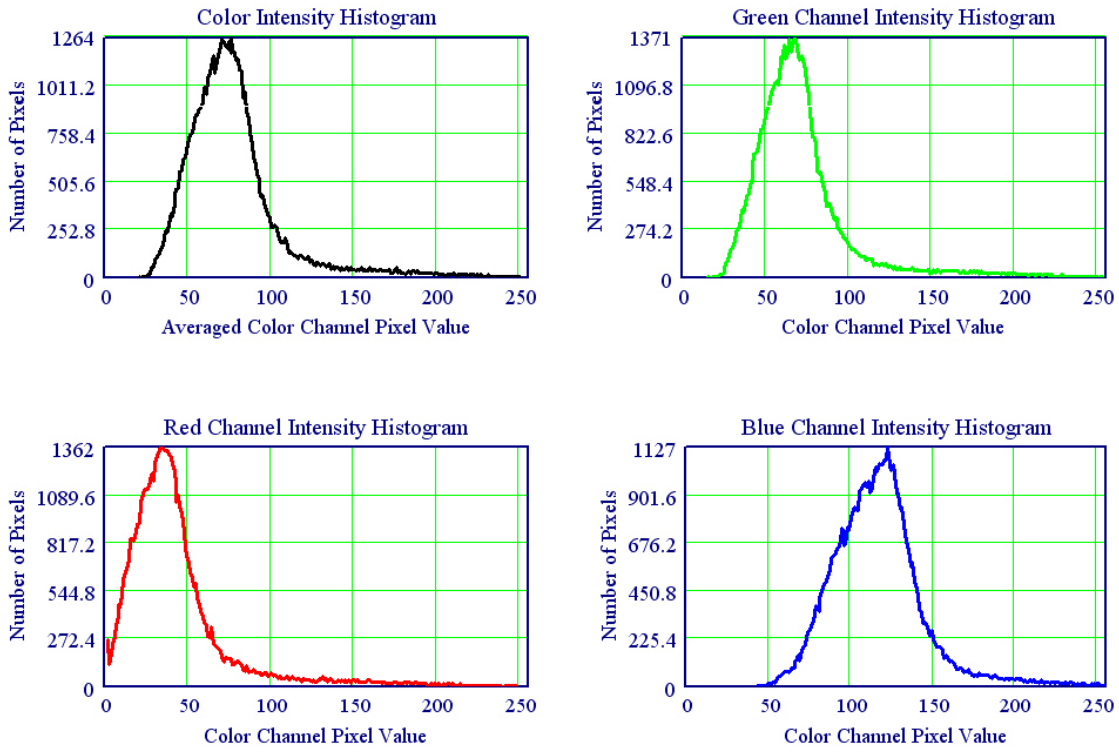

Figure 18: Color Histograms: Overall Image Intensity, Red Channel, Green Channel, and Blue Channel (from Top to Bottom) of Figure 17.

Figure 18 shows the histograms that correspond with the images in Figure 17. The x-axis of each graph is the integer value of the associated color channel. The y-axis is the number of pixels in the image that have that value. The top histogram is the average of all three color channels and therefore represents the overall intensity of each pixel. The information in these histograms corresponds to the way the color channel images appear.

The red channel is the darkest of the three, with a large portion of its pixels with a red value around 30 to 40. On average, the green is brighter than red in this image, having many of its pixel's green values in the 60 to 70 range. As expected, the blue has the highest average value of the three color channels, around 120 to 130.

The blue channel's distribution curve is wider than that of the red and green, which means the intensity distribution for the blue is more diverse than the others. This can be seen by inspection, but can also be determined by the maximum value of the curve. For blue, the maximum number of pixels that have the same intensity value is 1127. Red and green both have max values around 1370. Narrower color distributions must have more pixels compressed within a smaller area, causing the frequency of those pixels to be higher than for a wider band distribution. This phenomenon is not scientific fact and may not work for some cases, but for typical distributions like this one, it is a decent indicator for the relative width of the distribution.

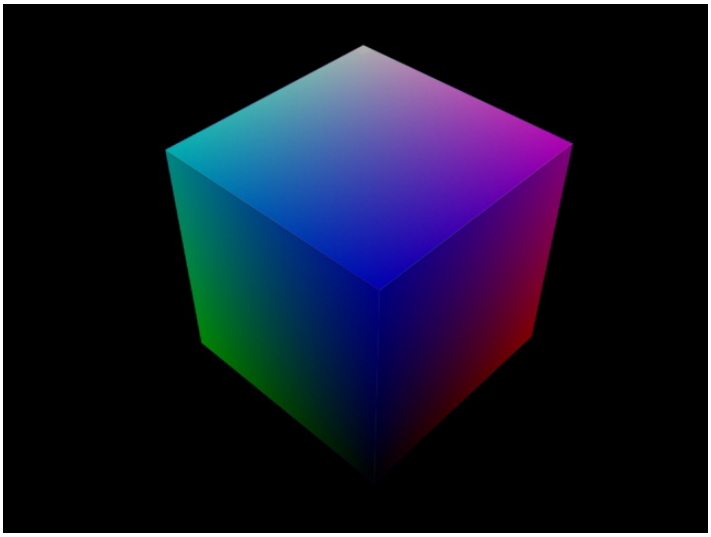

Figure 19: RGB Color Cube

Spatial Color Representation

Since computer color is in terms of 3 variables (red, green, and blue), it is useful to think of each color component as a spatial dimension. The result is a three dimensional representation of color information, where each data point represents the color of one pixel in the image. The spatial representation of color is referred to as RGB space.

Currently, a full color image is comprised of 24-bit pixels. This gives the red, green, and blue channels 8-bits each for representing color information. Therefore, each color component has a range of (0-255). The resulting space can be viewed as a color cube with a length, width, and height of 255 which contains all 16,777,216 possible colors, see Figure 19. By converting colors to coordinates, physical relationships can be found between colors that would not be apparent as just a series of numbers.

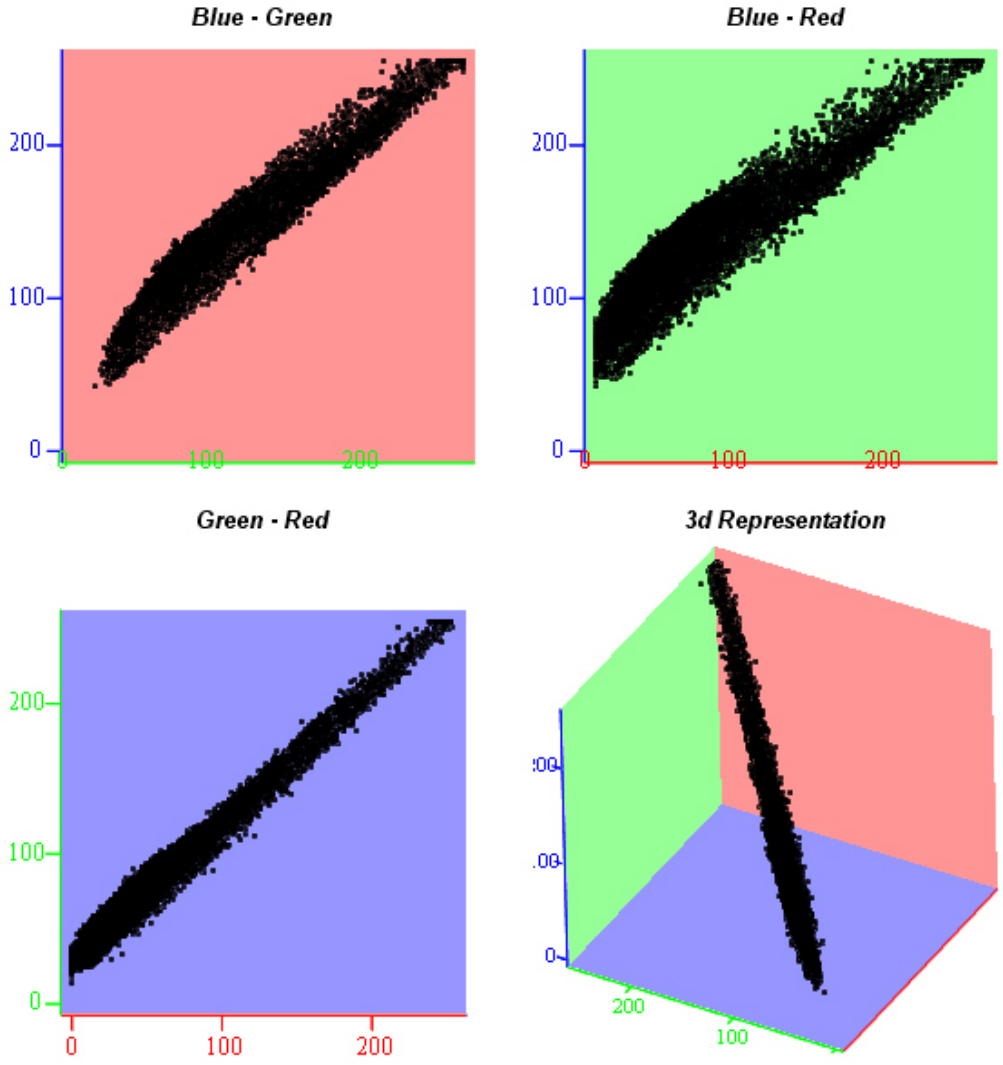

Figure 20: Image Color Representation in RGB Space for Figure 17.

The physical distribution of data for Figure 17, the sample image of the ocean, can be seen in Figure 20. A view of the data is shown along each axis, which allows for

correlations between each of the colors to be inspected. The color relationship between blue and green, blue and red, and green and red can be seen in the three flat graphs. The final graph of the data is a perspective view showing the layout of the pixels in RGB space. It is difficult to completely visualize the distribution without being able to rotate the graph and see the data from many angles, but the general idea should be clear.

From the top two graphs the relationship of the blue vs. red and green can be seen. In both instances the data is greater in the blue direction than the green or red, indicating the blue channel's dominance in the image. The bottom-left graph shows that the green is slightly more dominant than the red because of the upward shift in the data. These three graphs confirm what was shown in the color histograms. The final plot shows the overall distribution of data in RGB space. The data forms a fairly linear and tight distribution along the path from the black corner to the white corner of the color cube, with an offset to the blue direction. Most real-world images fall into this trend. There are very few extreme colors in the typical photo or video, such as, full intensity reds, greens, blues, cyans, etc. Most pixels will fall into these middle color ranges, often making separation of colors difficult.

Testing Methodology and Sample Images

There are several different methods of image processing used for this research because of the range of applications and situations that arose. As mentioned earlier, some statistical models work better in certain situations than others, so each method is shown and tested for a few different circumstances. The net worth of each classifier can be determined by the error it causes when used.

Classifier Error

Classification errors are two-fold for each class. An error occurs when a pixel is selected to be in a class when it does not belong, but an error also occurs when a pixel is not selected to a class when it does belong. The first type of error is referred to as a false-hit error; the second is called a missed-hit error. Each type of error has different effects on the functionality of the position system.

The first type of error, the false-hit, can cause more features to be found than exist. This error can lead to confusion in the position system algorithm and must be explicitly checked for. Although false features can usually be discounted, the process of checking multiple features for accuracy causes a significant speed problem. The second type of error, a missed-hit, is perhaps even more troublesome than the first. If a correct feature is missed completely, there is no information gathered from that image. In the case of the position system, this represents a condition where the vehicle's whereabouts are unknown. Obviously, this situation is undesirable and should be avoided whenever possible.

It is not usually possible to create a classifier that minimizes both of these types of errors. If a classifier is very narrow in its selection range, it is likely to minimize false hits but increase the number of missed hits. The opposite is true when a classifier is very wide. Often, it is desirable to minimize the sum of both types of errors, and get the minimum overall error. The best approach to use depends on the application. If the tracked vehicle is slow moving, then perhaps it may be beneficial to create a broad classifier that will always be able to find the vehicle and let the software take its time to discredit any false hits. A faster vehicle may need the program to process data at a higher frame rate, and therefore, must deal with the consequences of having a narrow classifier.

Testing Procedure and Error Calculation

The feature training data is found for each image by searching through it with an image editor and manually removing pixels belonging to the vehicle tracking feature. The groups of pixels that contain the feature are then placed into a separate image. Once this is done, the pixel values in the new image are tabulated and put into a feature data output file. The original image with the feature pixels removed is then inspected. The data from this file is put into a separate, non-feature, output file. Each image in the set is handled in the same way and the data is appended to the existing output files. The end result is a file that contains the feature pixel information from all of the feature data input files and another file that contains the non-feature, or outside, pixel data from all of the outside files.

The feature data can now be used to create a statistical classifier. Once this classifier has been defined, it must be checked against the data used to create it. The missed-hit error will be defined by:

$$E_m = \frac{N_{missed}}{N_{feature}} \tag{36}$$

Where E_m is the percentage of feature pixels missed by the classifier when tested against the training data. N_{missed} represents the number of feature pixels missed by the classifier. $N_{feature}$ is the total number of pixels in the feature training data file.

The false-hit error can be found in the same way by inspecting the pixels in the outside information data file. For the false-hit error:

$$E_f = \frac{N_{hits}}{N_{outside}} \tag{37}$$

Where E_f is the percentage of false hits caused by the classifier. N_{hits} is the number of pixels found by the classifier in the non-feature data file. $N_{outside}$ represents the total number of pixels in the non-feature data file. All of this information is determined automatically by the training data generation program (for more information, see Appendix C).

If the overall error is desired, it can be found from the sum of the false-hit and missed-hit errors:

$$E_{tot} = E_m + E_f \tag{38}$$

Where E_{tot} is the overall error. The overall error is calculated by summing the two averages instead of summing the number of incorrect hits and dividing by the total number of pixels. This is done because the number of pixels in the outside data file will usually greatly outnumber the pixels in the feature file. Using this data directly to calculate the overall error would cause a significant bias toward the false-hit error.

Test Images

The test images span a range of scenarios that show the strengths and weaknesses of each pattern classification method. Each set of images is comprised of four different photos. Each photo shows a different position, point-of-view, or lighting condition in an attempt to create more a realistic classifier. Of course, using only four images to train each classifier is not very robust but was necessary to fit within the space constraints of this document.

Each set of images is shown with the original image on the left, the feature image in the middle, and the non-feature image on the right. The feature image displays the

groups of feature pixels on a white background. The outside data is shown with the feature locations replaced with white pixels as well. When processing these images, pure white pixels are assumed to be null areas of the image and ignored so they are not added into the color distribution.

It's inappropriate to use a training data image to test the classifier that it has trained. Instead, a separate image needs to be used that will properly show the tracking feature, but is different enough to show that the classifier works on non-feature data. The test image for the classifier is shown underneath the group of training images. Each classifier is tested on a few of these images with the detected pixels marked in a different color.

In addition to the images, the color histograms and RGB color plots are shown of the data. The distributions that are shown have been re-sampled so that the data for the feature and the outside data are approximately equal in proportion. This was necessary because of the extreme amount of non-feature data contained in the outside images. The original data pulled from these images runs from 1,000,000 to 10,000,000 data points. This amount of data is a real strain on the mathematic software packages that are used to create graphs for this document. To compensate for this, the outside data sets were down-sampled to approximately 100,000 data points and the feature data sets were up-sampled to around the same size. The resulting density plots change very little for re-sampled data. Data re-sampling techniques are discussed along with their associated errors in Appendix D.

The histograms for these images show both the feature data and the non-feature data plotted together. The feature data is represented by a solid line that is the color of

the color channel represented in the histogram. The non-feature data is represented by a black dashed line. This is done so that a comparison of the feature data and the non-feature data can be made for each color channel. In addition to the color channel histograms, data point plots in RGB space are shown as well. The feature data points are shown in white and the non-feature data points are shown in black.

Test image set 1: Road lines

Training Images:	Figure 21 (Page 62)
Test Image:	Figure 22 (Page 62)
Color Histograms:	Figure 23 (Page 63)
RGB Space Plot:	Figure 24 (Page 64)

Test image set #1 is footage from a street on the University of Florida campus. This video was shot to use for visual road line recognition. The video was taken with a digital video camera while walking down the street. The four images span approximately 45 seconds of video footage.

The road lines along with the biking lane symbols have been separated from the rest of the image to use as features. This is a difficult image to process because of the minimal contrast between the selected features and the rest of the image. The sky reflecting off the road in the distance is a significant problem because it makes the road appear to be the same shade of white that the road lines are. In addition to the road reflections, the sky itself is overcast and appears to be similar to the color of the road lines.

Test image set 2: Yellow lid

Training Images:	Figure 25 (Page 65)
Test Image:	Figure 26 (Page 65)
Color Histograms:	Figure 27 (Page 66)
RGB Space Plot:	Figure 28 (Page 67)

Test image set #2 are pictures of a lid to a container placed in several places around the author's apartment. These images were taken with an inexpensive digital camera in different lighting conditions for each image. These images were taken solely for the purpose of testing the different image classification techniques.

The main difficulties of detecting the yellow lid are the inconsistency of the lighting conditions and the poor quality of the images. This was done intentionally to show the effects of poor image quality on each of these classifiers. The color of the tracked feature is fairly unique in each of the images despite the poor photo quality so the detection is not overly difficult.

Test image set 3: Simulated warehouse:

Training Images:	Figure 29 (Page 68)
Test Image:	Figure 30 (Page 68)
Color Histograms:	Figure 31 (Page 69)
RGB Space Plot:	Figure 32 (Page 70)

Test image set #3 is an animation of a small warehouse. These images were created and rendered using a 3D modeling package. This set of images was created for testing the image classification techniques as well. Several lights were added to the scene to create different lighting effects from different camera views.

Simulated images have been an area of particular difficulty for some of the image classifiers. Many simulations were done for this research since it is much quicker and easier than taking real footage for many of the environmental scenarios. Simulated images do not tend to have the color scattering that is present in real photos. Therefore, when light reflects off of a surface in a simulation, a nice and consistent transition of colors can be seen. At times, even large areas can have the exact same pixel color value. Because of this data tends to cluster into globs with simulated images. Statistical models,

which are dependent on the typical scattering of data, tend to be highly inaccurate with simulations.

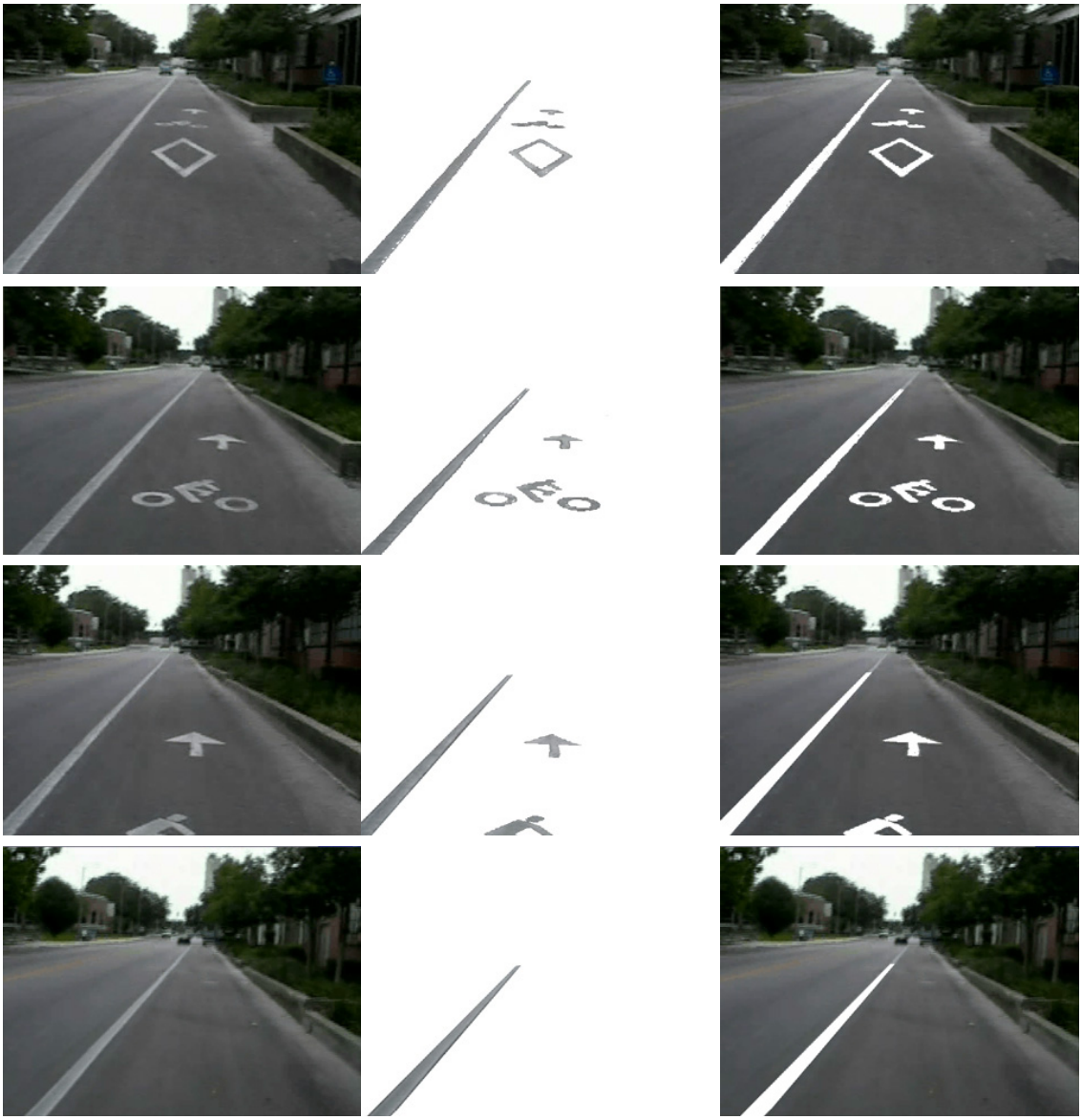

Figure 21: Image Set 1 Training Data: Road Lines. Left Column Shows Original Images. Middle Column Shows Feature Pixels. Right Column Shows Non-Feature Pixels.

Figure 22: Image Set 1 Test Image

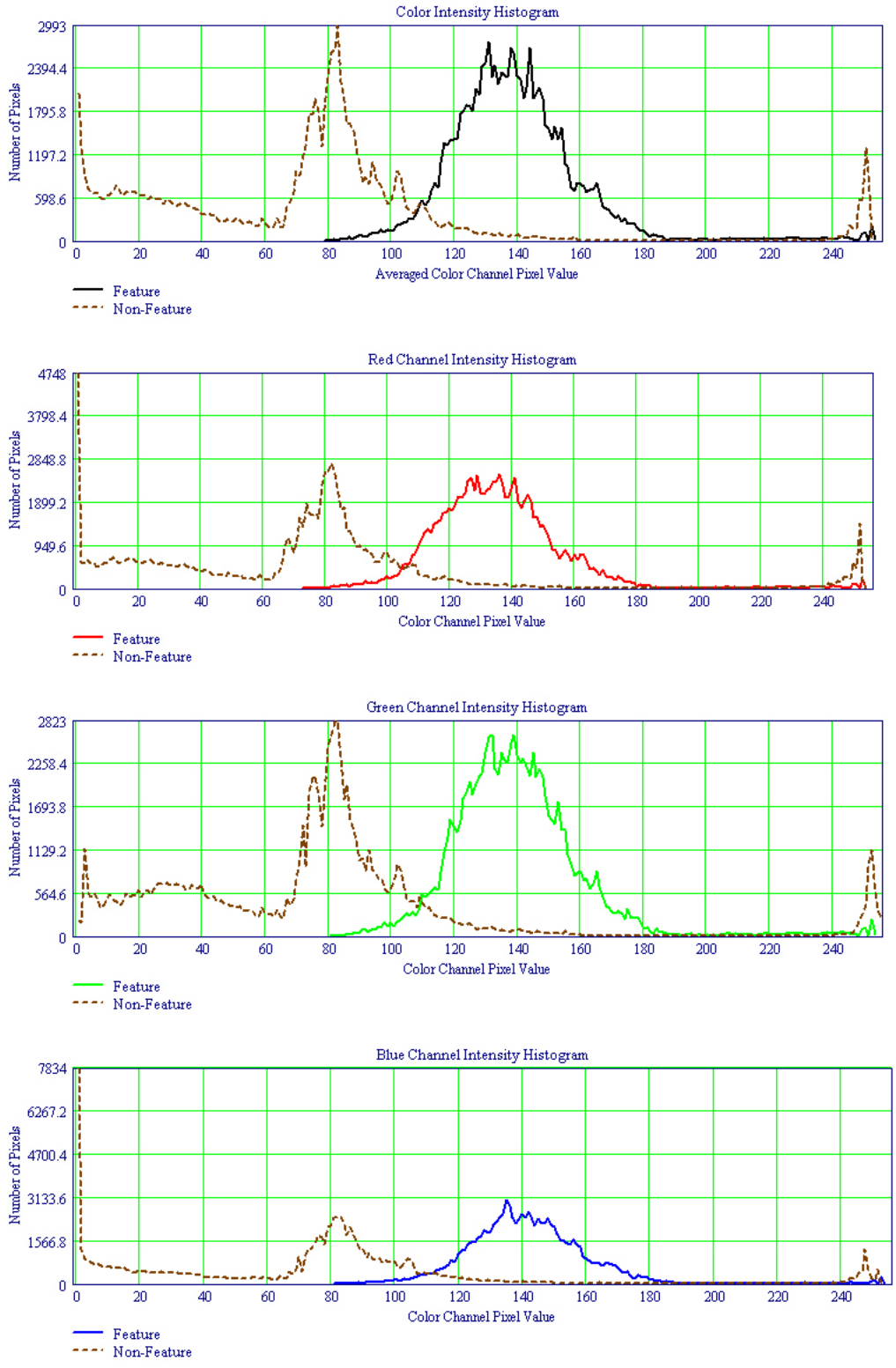

Figure 23: Image Set 1 Training Data Color Histograms.

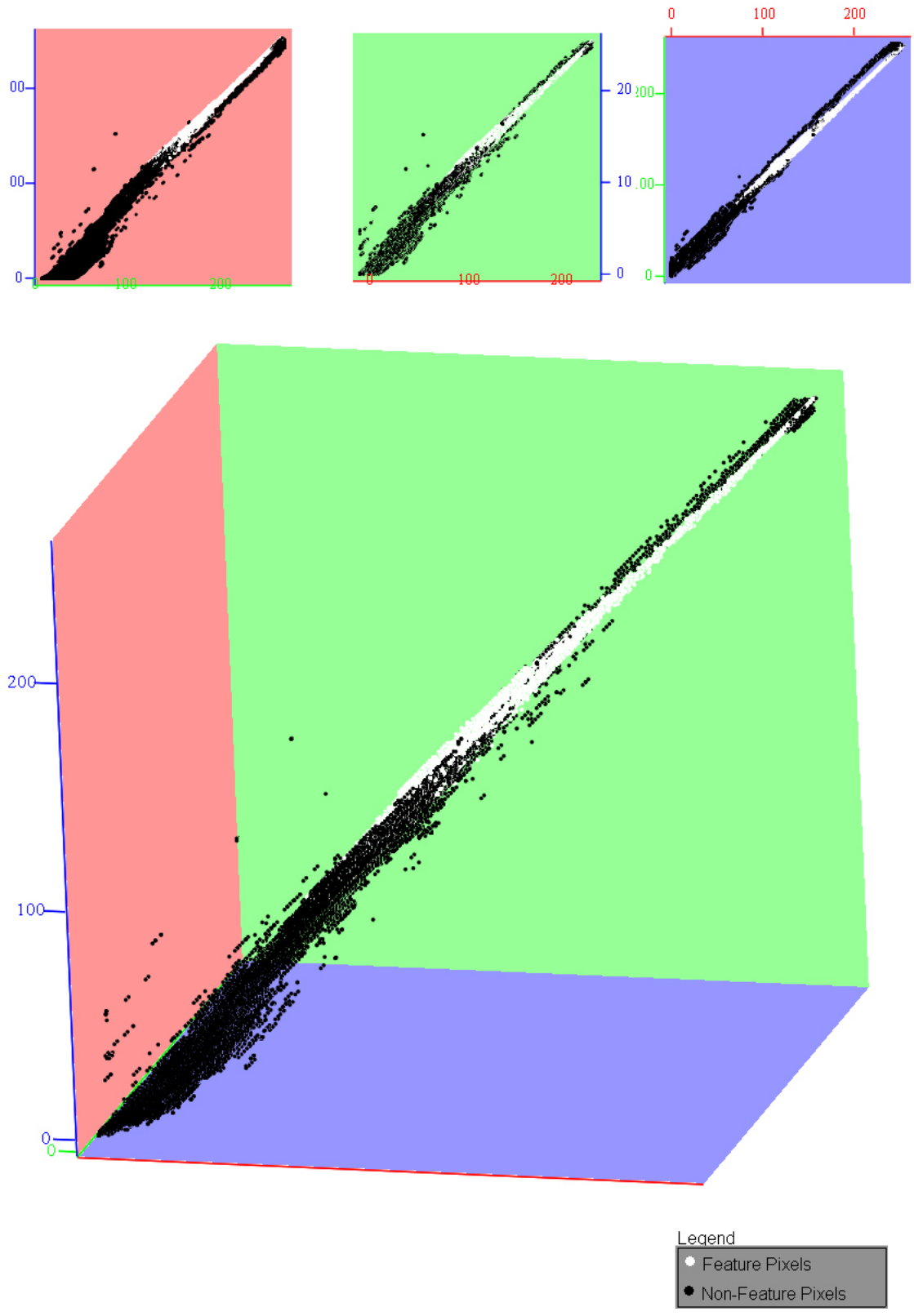

Figure 24: Image Set 1 Training Data in RGB Space.

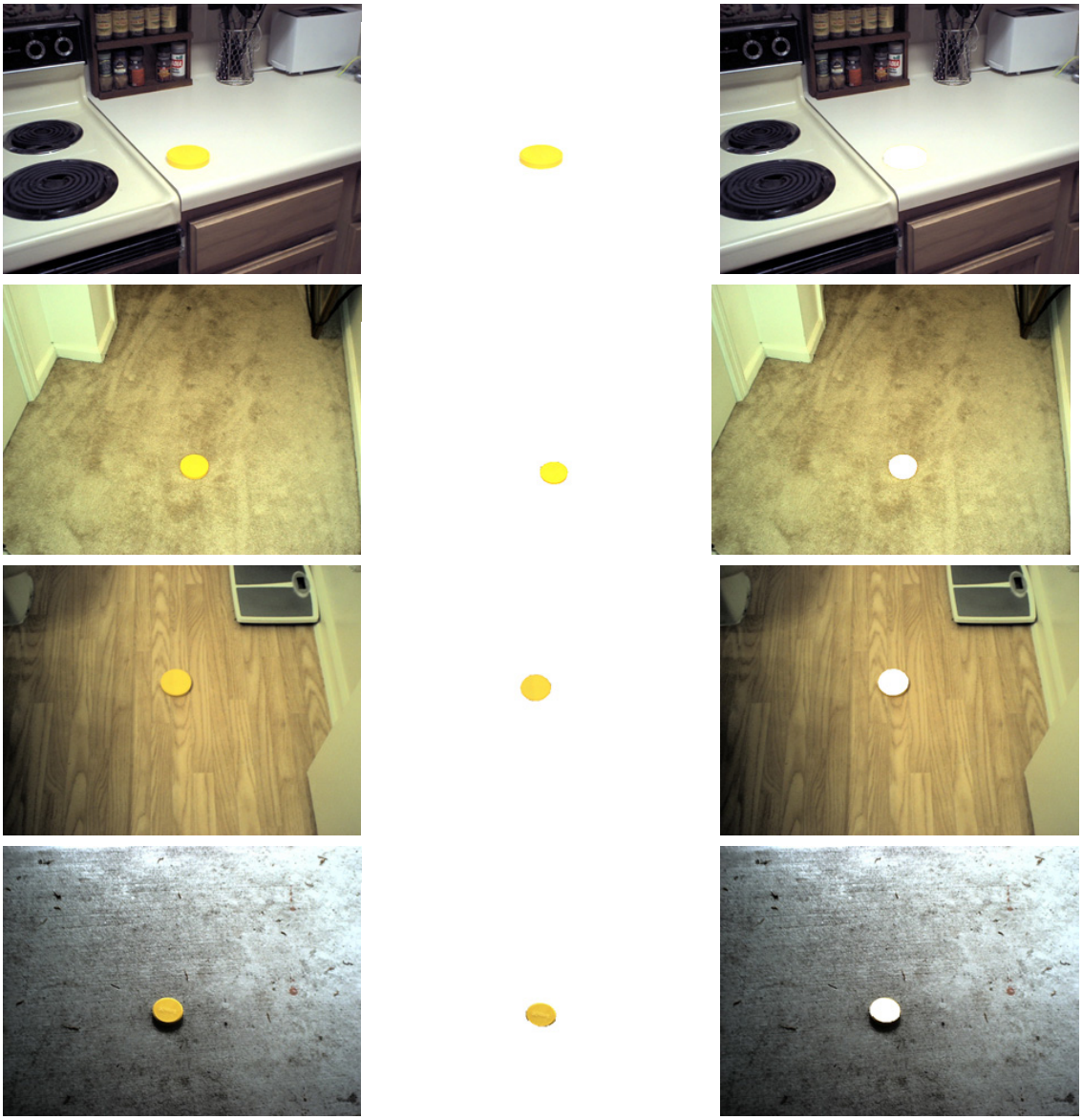

Figure 25: Image Set 2 Training Data: Yellow Lid. Left Column Shows Original Images. Middle Column Shows Feature Pixels. Right Column Shows Non-Feature Pixels.

Figure 26: Image Set 2 Test Image

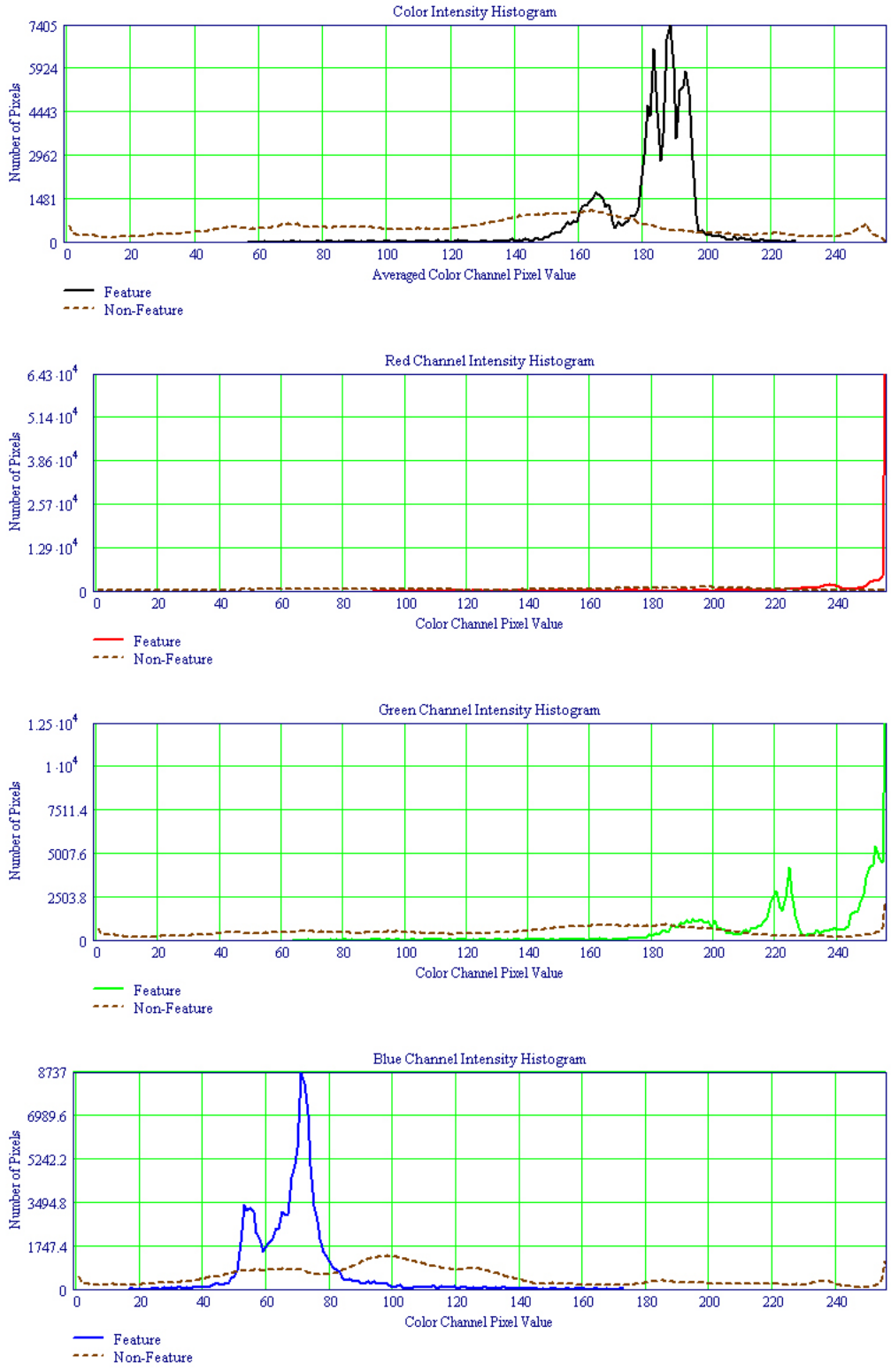

Figure 27: Image Set 2 Training Data Color Histograms.

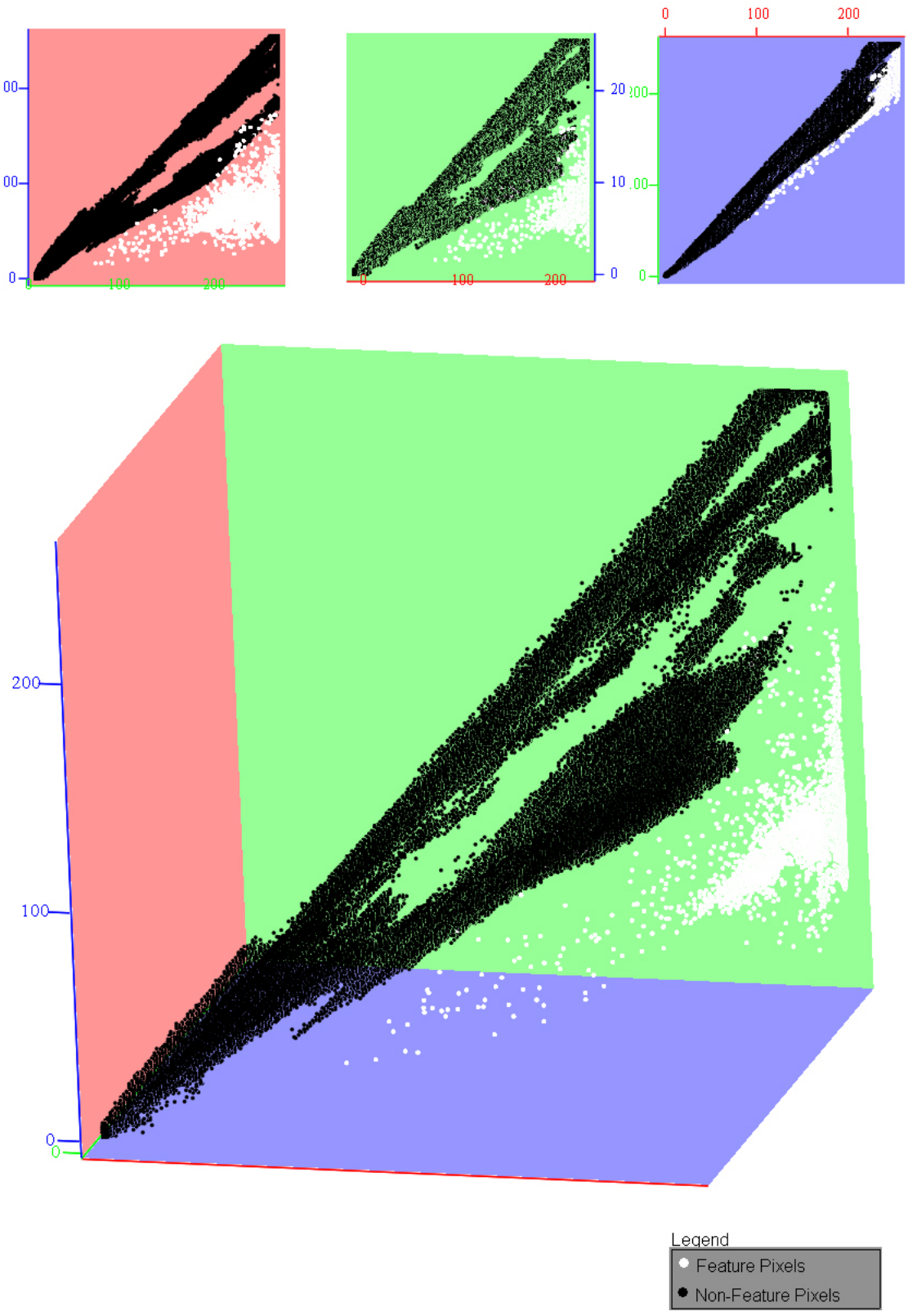

Figure 28: Image Set 2 Training Data in RGB Space.

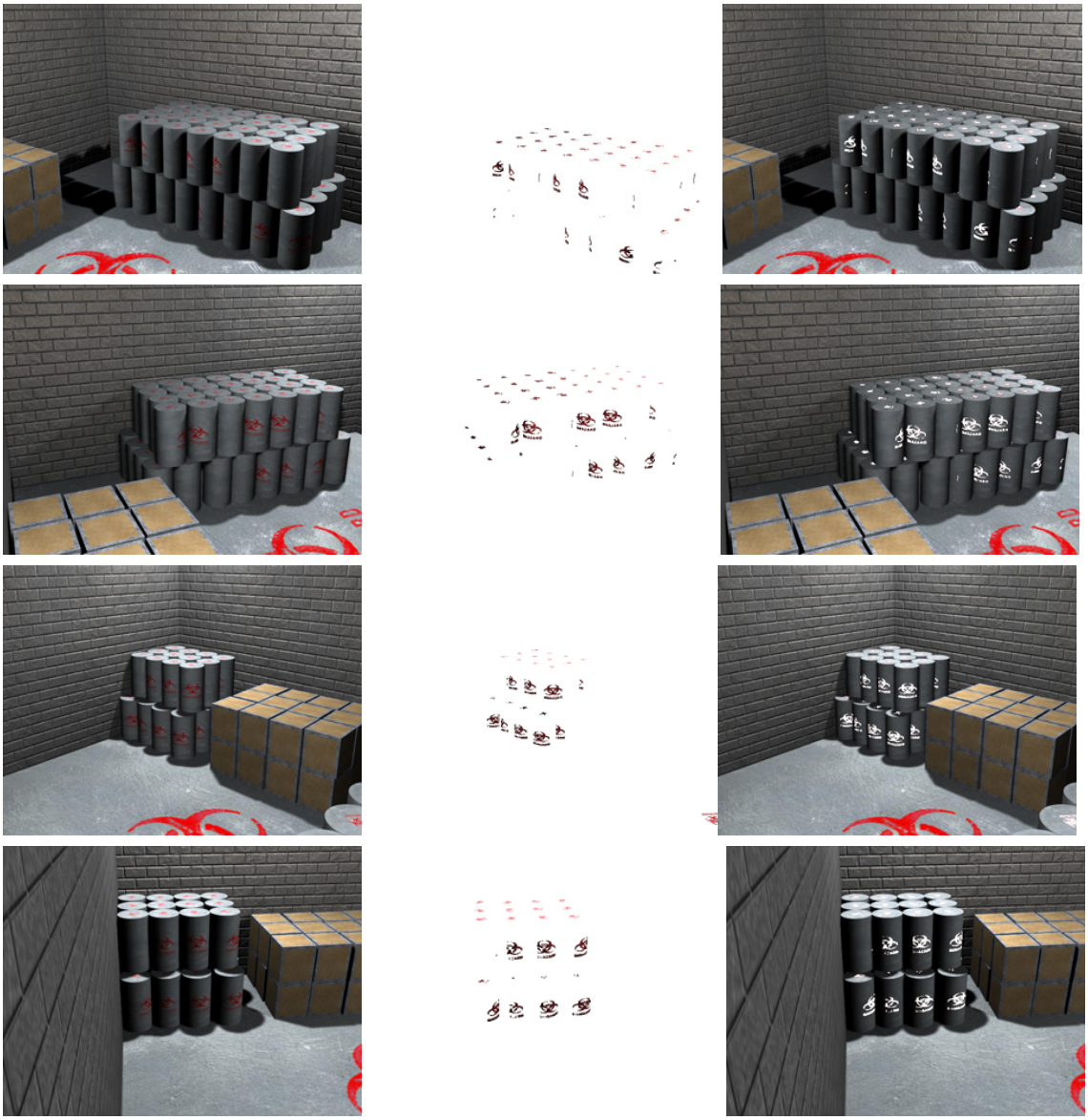

Figure 29: Image Set 3 Training Data: Simulated Warehouse. Left Column Shows Original Images. Middle Column Shows Feature Pixels. Right Column Shows Non-Feature Pixels.

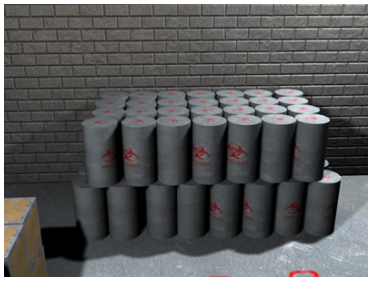

Figure 30: Image Set 3 Test Image

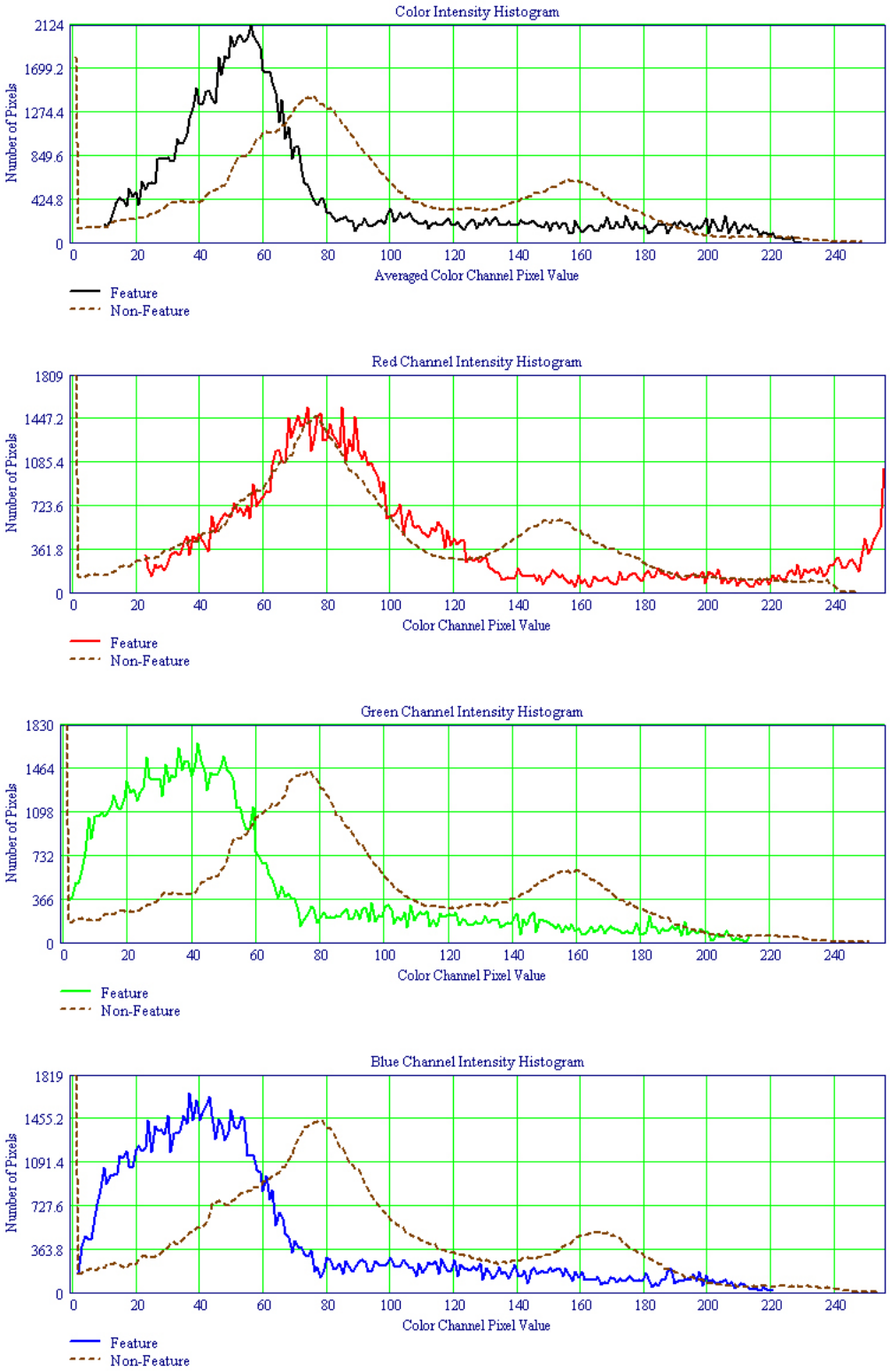

Figure 31: Image Set 3 Training Data Color Histograms.

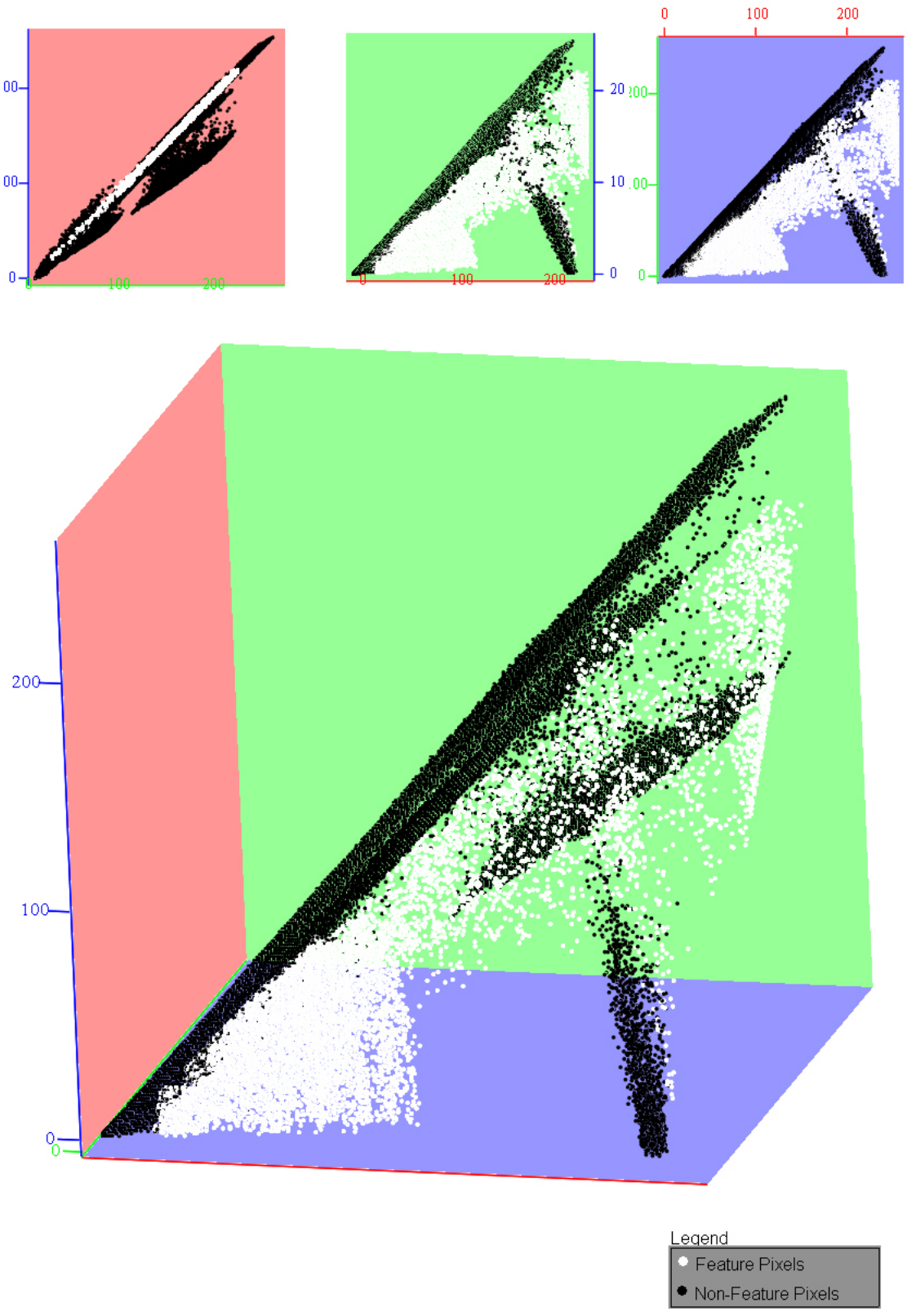

Figure 32: Image Set 3 Training Data in RGB Space.

Statistical Image Classifiers

Color Range

The least complicated of the statistical classifiers is color range. This entails determining a range of colors to select for each color channel separately. This method is simple and fast. If the feature consistently resides in a significantly different region of RGB space than the rest of the image, this method works very well. Unfortunately, it is rare to find a case that fits this profile.

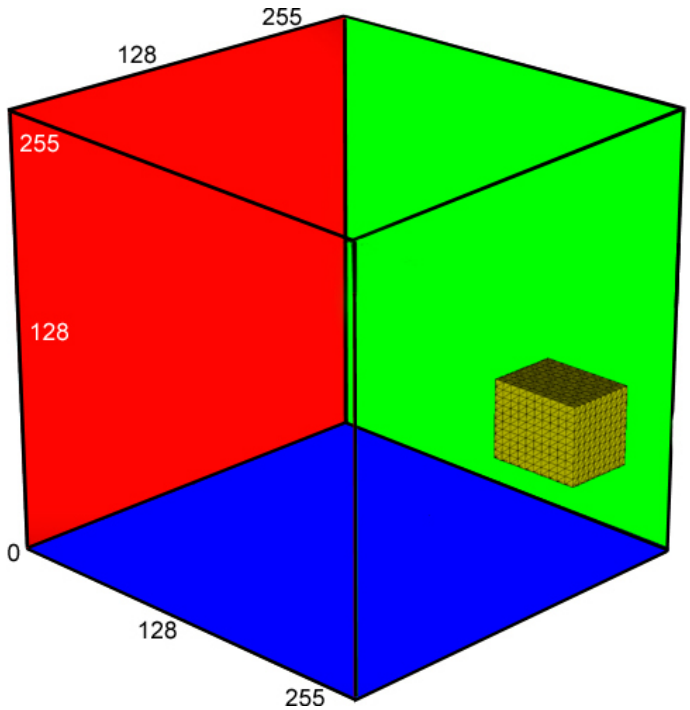

Figure 33: Color Range Distribution Profile in RGB Space

Defining a color range essentially determines a box in RGB space, in which every pixel is selected. This box is always oriented with the R, G, and B axes and allows for very little fine-tuning, see Figure 33. An acceptable color range can be determined by looking at the histograms of the color channels, but the practicality of the system is better found by inspecting the data in RGB space and determining if a block of the feature data can reasonably be separated from the rest of the outside data.

Mathematics

The calculations for this classifier are quite simple. The name alludes to exactly what is needed, a range of colors to search for. For every color channel, a low and high

value must be specified. These values can be any integer inside the color cube, and the range can be as large as desired. Caution should be used in defining oversized ranges because the amount of error can become vast as the dimensions of the classifier increase.

Choosing an acceptable range to use will depend on the application, but it is always a good idea to view the training data in RGB space. This will make it easier to determine the portion of space that the feature occupies and will show if any outside data is likely to interfere. Also, checking the color channel histograms can show the distribution of the feature color with respect to the other colors in the image.

One way to determine a feature range is to create a Gaussian model of the data for each color channel. Gaussian distributions, also known as normal distributions, are perhaps the most common distributions in nature. Many systems in nature become Gaussian the amount of data samples increases. A large data set is intrinsic to having a normal distribution properly defined. Images, therefore, are well suited for use with a Gaussian model because the sample sizes usually run from tens of thousands to millions. A one dimensional, or univariate, Gaussian is defined by:

$$p(x) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left[-\frac{1}{2}\left(\frac{x-\mu}{\sigma} \right)^2 \right] \tag{39}$$

where *p(x)* is the Gaussian probability density function (PDF) for the variable *x*. The mean of the system is represented by μ and the standard deviation is represented by σ. The PDF can be completely described by these two quantities. A Gaussian distribution has approximately 68% of its data points between $\mu - \sigma$ and $\mu + \sigma$, 95% of its data points between $\mu - 2\sigma$ and $\mu + 2\sigma$, and 99.7% of its data points between

$\mu - 3\sigma$ and $\mu + 3\sigma$, see Figure 34 [Dud01]. With this in mind, a good starting range is between $\mu - 2\sigma$ and $\mu + 2\sigma$. This will be referred to as the two-sigma rule.

This principle can be used to determine the approximate distribution of the data for each color channel separately. This information can be used to make an informed decision on where to put the decision boundaries. Unfortunately, the univariate Gaussian distribution does not take the distribution of the non-feature data into consideration. Finding the best decision boundary for the color range model will require that error reduction be done for several different deviation amounts to see where the minimal classifier error is.

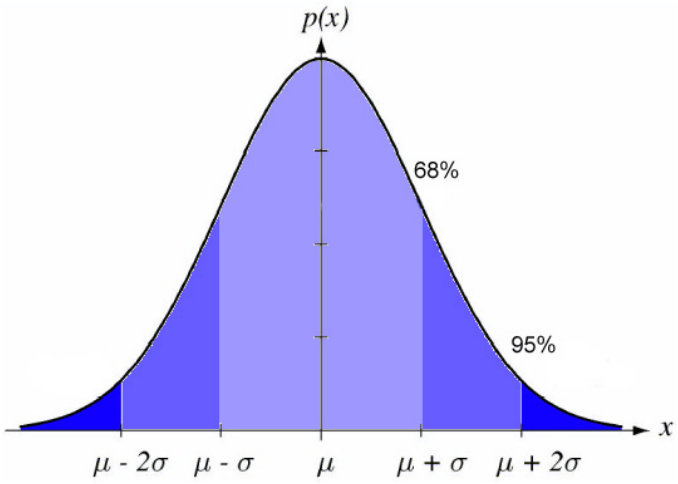

Figure 34: Univariate Gaussian Distribution Curve

Test on image set #2

A good case to use this classifier would be image set #2, the yellow lid. By looking at the layout of the pixels in RGB space in Figure 28 on page 67, it can be seen that the yellow pixels are separated very well from the non-yellows. A univariate Gaussian was created for each of the color channels, see Figure 35. The Gaussian distribution has been added to the original histograms and is represented by the brown dashed line. The $\mu \pm 2\sigma$ range was used for each color channel and yielded the following color ranges:

Red: 221 – 255

Green: 176 – 255

Blue: 42 – 96

The result of this classification can be seen in Figure 35. The outside pixels that are properly classified are shown in black, the feature pixels that are properly classified are shown in white, and the missed-hit pixels are shown in blue. There are no false-hit pixels resulting from this classifier on the training data.

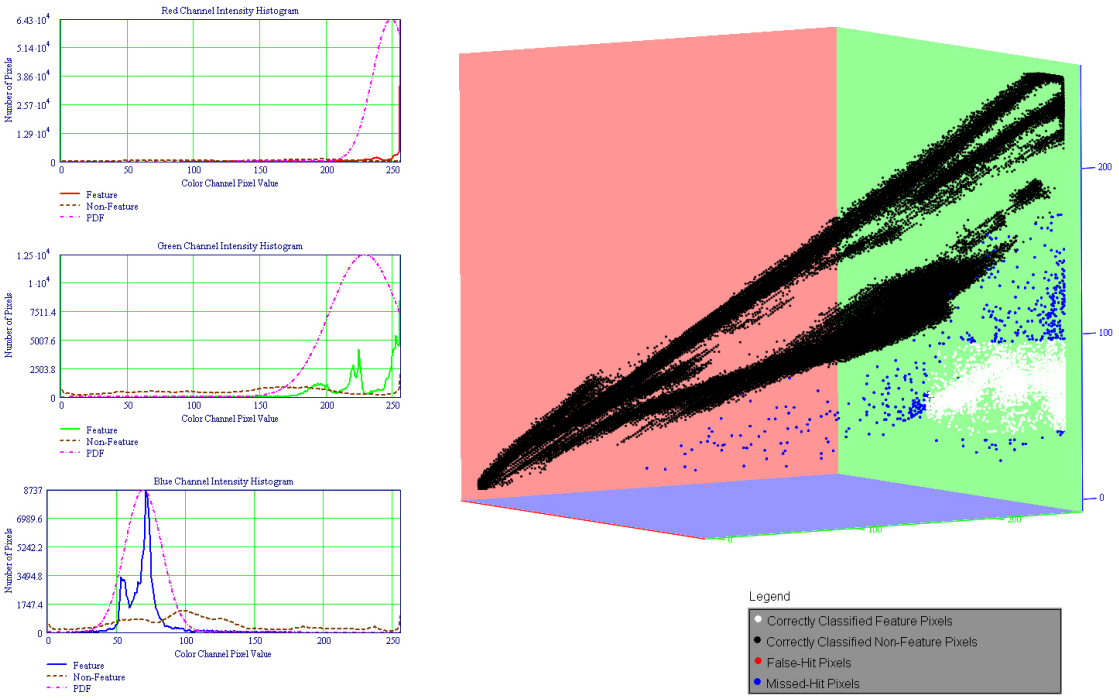

Figure 35: Yellow Lid Histograms with Univariate Gaussian Overlay (Left). Classifier Results Shown in RGB Space (Right).

This classification is not bad, considering the inelegance of the algorithm. The missed-hit error on the feature is 5.9% while the false-hit error is non-existent. The overall error is, therefore, 5.9%. This result is surprisingly good for this particular test case.

This classifier was tested on the sample image from Figure 26. In this image the lid is in a placed in a location completely separate from any of the training images. The

pixels that are classified to be feature pixels are highlighted in red. The color range classifier works pretty well on this data set by finding a good portion of the pixels containing the lid and causing no false hits, see Figure 36.

Color range may not create the best classification for this class, but it is not bad. In some cases the increase in processing speed acquired by using a low-level method like this may be more beneficial than a slower, more robust method.

Figure 36: Yellow Lid Image Processed with Color Range Classifier. Classifier Hits are Shown in Red.

Test on image set #1

Image set #1 is a terrible candidate for this type of classifier. While the road lines are a difficult feature to process for any method, it is a worst-case-scenario for the color range detection method. The feature data is very comparable to the outside data. The feature data runs in a narrow path along with the non-feature data. Using color range on this case yields significant errors and is highly impractical.

Again, a Gaussian distribution was created for each color channel and the two-sigma rule was used to determine range values. For this case they turned out to be:

Red: 92 – 179

Green: 98 – 181

Blue: 101 – 183

Figure 37 shows the road line data Gaussians and RGB space plot of the classifier results. Again, white is correctly classified feature data, black is correctly classified

outside data, red represents false-hits, and blue represents missed-hits. The false-hit error is not very good at 12.5% for the training data, but the missed-hit error is acceptable at 3.5%. Therefore, this classifier gives an overall error of about 16%. This classifier, while getting a good portion of the feature pixels, gets a large portion of non-feature pixels and would not be advisable for this data set.

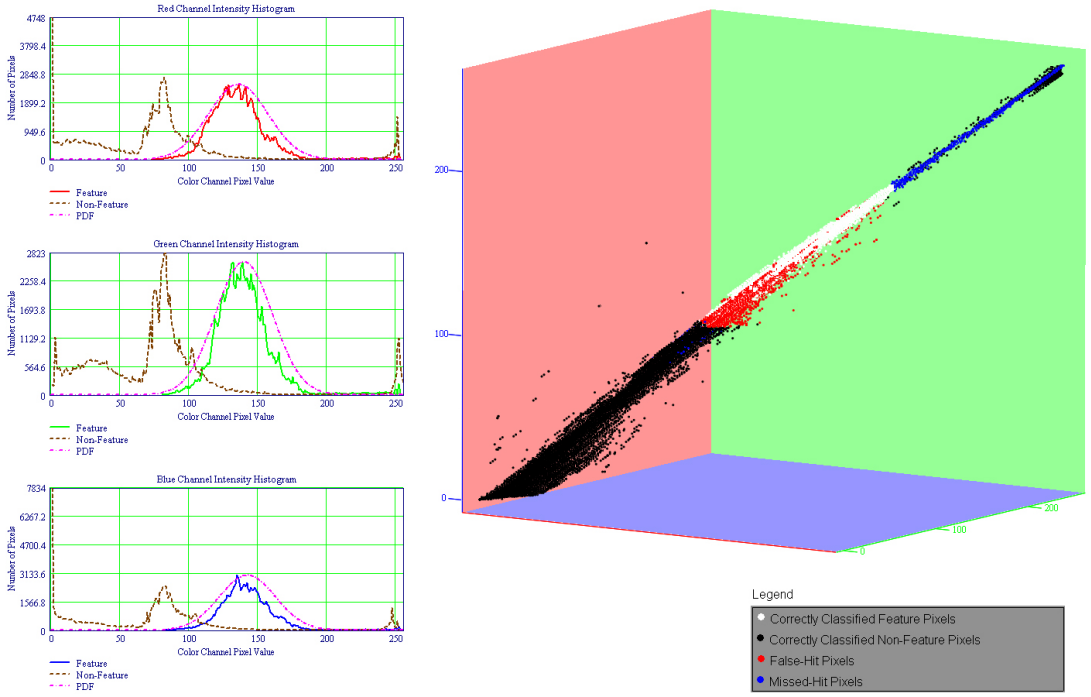

Figure 37: Road Line Histograms with Univariate Gaussian Overlay (Left). Classifier Results Shown in RGB Space (Right).

This classifier was run on the test image from the image set, shown in Figure 22. The classifier is able to capture all of the road lines in the image, but the part of the street that is illuminated by the reflection of the sky causes difficulties. Practically this whole portion of the street was mis-classified to be a road line. The result can also be seen in Figure 38, with the expected road line pixels shown in yellow.

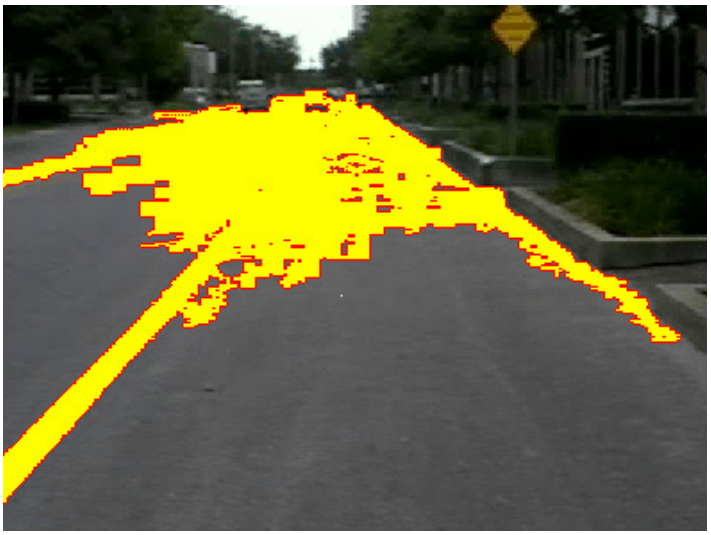

Figure 38: Road Lines Image Processed with Color Range Classifier. Classifier Hits are Shown in Yellow.

Color Direction

The color direction classifier is useful in situations where lighting conditions are inconsistent. Instead of using low and high pixel color values to define a classifier, the vector direction of a target color in RGB space is used along with a range in the three color directions. This specifies a conic section in RGB space that starts from the coordinate origin and extends to the bound of the RGB cube. Everything inside this section will be considered the target feature, see Figure 39.

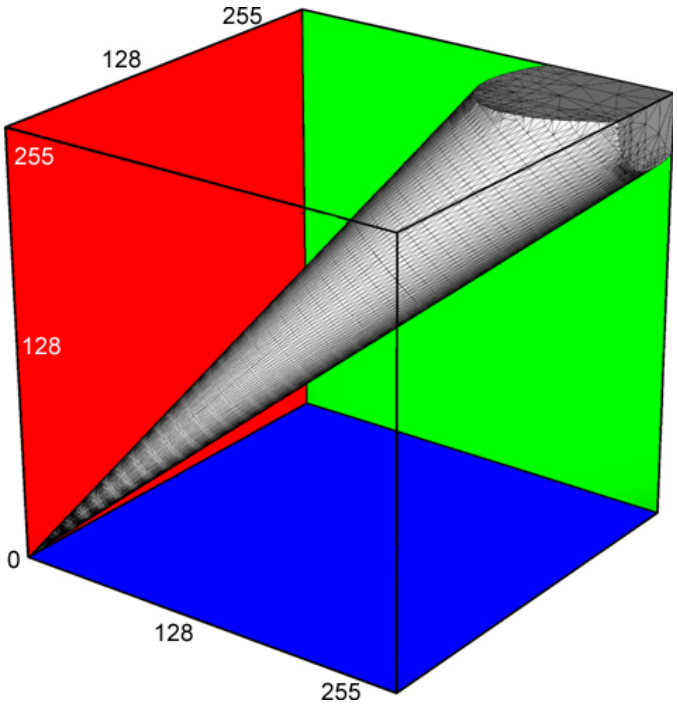

Figure 39: Color Direction Distribution Profile in RGB Space

The advantage to using color direction over color range is that the intensity information, or brightness, is removed from the color distribution. Therefore, a dark color of green will appear to be the same as a lighter shade of green. This allows for variance in the target color due to light conditions and shadows.

Unfortunately, the color direction classifier requires that each pixel inspected be normalized, which requires a good amount of processing time (The normalization process is discussed in the next section). In situations where the lighting conditions change frequently in the image, color direction offers superior adaptability when tracking a feature.

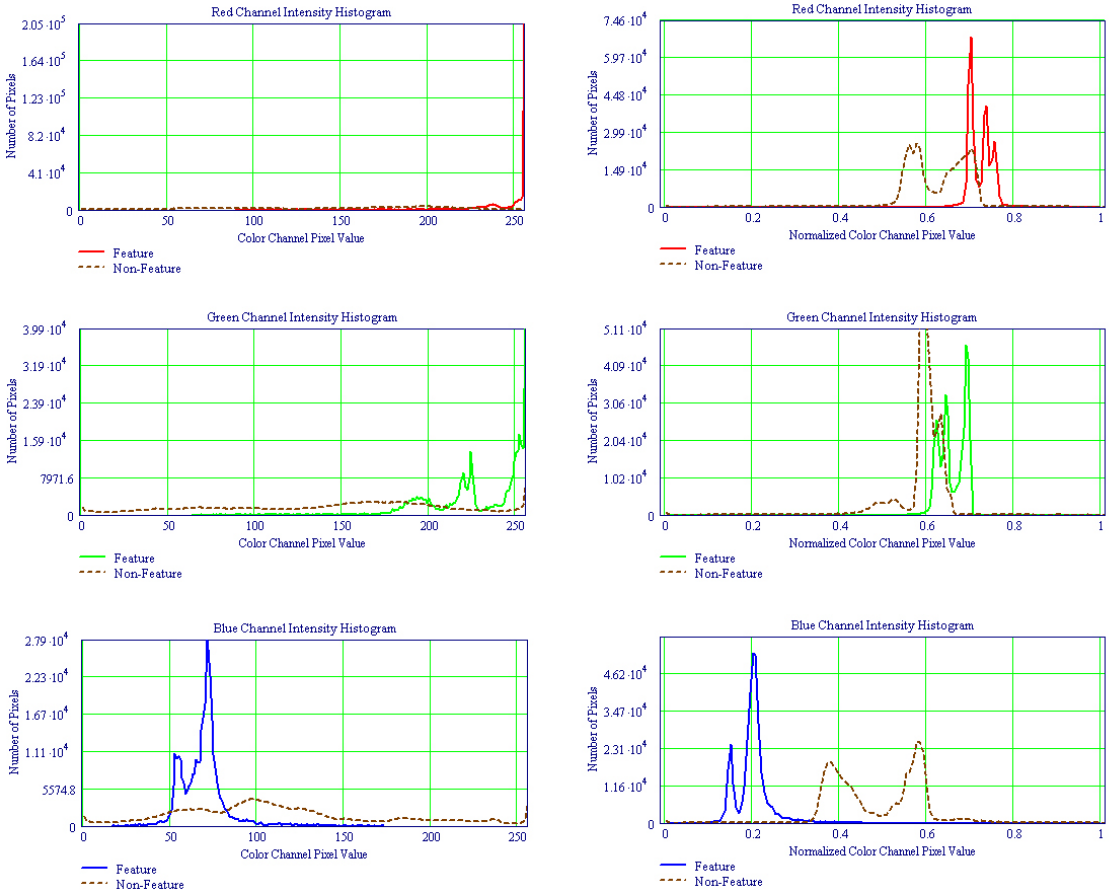

Figure 40: Yellow-Lid Data. Color Histograms (Left). Color Direction Histograms (Right).

Mathematics

To use the color direction classifier, a vector must be made in RGB space that runs from the origin to the feature color's coordinate. This vector then must be normalized to get the directional component of the vector. A range in the red, green, and blue directions must be specified to create the conic section. This information can once

again be obtained by the histogram and the univariate Gaussian model of the data. This time, though, the data represents the vector directional components instead of actual red, green, and blue pixel values, see

Figure 40. The color direction histograms give the normalized direction of each pixel in the image. A color direction value of 1 in any color channel means that the pixel is comprised of only that color. A value of 0, therefore, represents none of that color is present in the selected pixel. The normalized pixel direction vector can be calculated by the equations:

$$\hat{R} = \frac{red}{\sqrt{red^2 + grn^2 + blu^2}}$$
$$\hat{G} = \frac{grn}{\sqrt{red^2 + grn^2 + blu^2}} \quad (40)$$
$$\hat{B} = \frac{blu}{\sqrt{red^2 + grn^2 + blu^2}}$$

Where $\hat{R}$, $\hat{G}$, and $\hat{B}$ are the coordinates of the unit vector direction of the pixel point in RGB space.

Along with the color direction vector, a value range must be defined in each of the color directions. The shape of the conic section will change with respect to the range values that are selected. The shape of the conic can only change in the two directions perpendicular to the mean vector. This means the range values will affect the shape of the conic in different ways depending on the direction of the mean vector. For instance, if the mean vector is pointing directly in the direction of the red axis, the red range value will have no effect on the shape of the conic.

Test on image set #3

This simulation of a warehouse is typical of the images used to test this position system. Unfortunately, complications arise from using simulated images because of the orderly fashion that image colors are distributed with lighting changes. Some of the more sophisticated algorithms have a difficult time properly classifying data because of data clustering that occurs from non-realistic lighting conditions.

This problem was the reason the color range classifier was developed. This classifier allows a certain color to be chosen and tracked regardless of shadow conditions in the image. The exact color that is used to define a feature in a simulation model can be entered as the mean vector of the color range classifier. With a minimal variance set for each color, the features can be found reliably without over-classifying the data. This makes image set #3 a great example for the use of the color range classifier.

This image's normalized color distribution does not lend itself well to the univariate Gaussian distribution. The red channel is close enough to a Gaussian for the distribution to work, but the green and blue exhibit a ramping effect that creates a shifted Gaussian curve, see Figure 41. So instead of using the two-sigma rule on the Gaussian curve to find an acceptable range for the classifier, the process was done by inspection.

All three color channel normalized distributions have apparent falloff locations. These locations were used as high and low ranges. From these high and lows for each color channel the direction can be found by:

$$\hat{R} = \frac{redHigh + redLow}{2}$$
$$\hat{G} = \frac{grnHigh + grnLow}{2} \tag{41}$$

$$\hat{B} = \frac{bluHigh + bluLow}{2}$$

And the variance can be found by:

$$redVar = \frac{redHigh - redLow}{2}$$
$$grnVar = \frac{grnHigh - grnLow}{2} \quad (42)$$
$$bluVar = \frac{bluHigh - bluLow}{2}$$

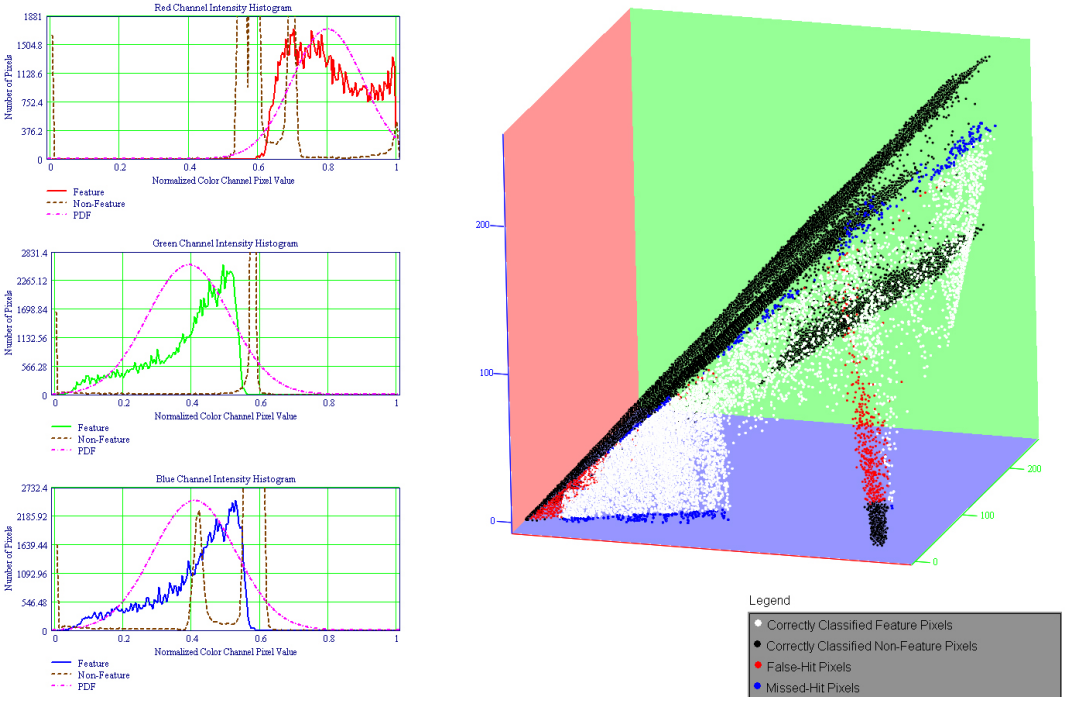

Figure 41: Simulation Warehouse Normalized Histograms with Univariate Gaussian Overlay (Left). Classifier Results Shown in RGB Space (Right).

For this data set, the range values used were:

$$Red = .61 - 1.0$$

$$Grn = .05 - .55$$

$$Blu = .10 - .55$$

This accounts for a large percentage of the feature pixels with minimal errors. The errors incurred are only 1.4% error for false hits and 4.5% for missed hits. Therefore, the overall error for this classifier is 5.9%, which is not too bad. The classifier results in RGB space can be seen in Figure 41.

Figure 42: Warehouse Image Processed with Color Direction Classifier. Classifier Hits are Shown in Yellow.

The color range classifier was run on the warehouse test image (shown in Figure 30). Overall, this classifier works the best for the simulate warehouse images. The biohazard logos on the sides of the barrels are found effectively. The logos on the tops of the barrels escape detection, but the false hits are negligible.

Test on image set #1

The road line data is a problematic image set for this classifier for many of the same reasons it was difficult for the color range classifier. The feature data is intermixed with the non-feature data in a respect that makes it very difficult to separate. For this classifier in particular, the data causes a secondary problem. The road lines that are being searched for are a dark white or a light shade of grey. Most of the ambient pixels, such as the asphalt, sidewalks, and the sky are also varying shades of grey. This classifier, by its definition, attempts to group different shades of the same color together to account for

shadowing. This makes the classifier think that the asphalt, sidewalks, sky, and road lines are all a similar color.

Deciding the direction and variance of this distribution was not difficult. Figure 43 shows the normalized color direction distributions. The red, green, and blue follow an almost perfect Gaussian distribution. In fact, the distribution is so close to the actual data that it is difficult to see the Gaussian curve. It is also apparent from this figure that the non-feature data resides in almost the exact same location as the feature data. Obviously, this makes the feature data difficult to extract from the image.

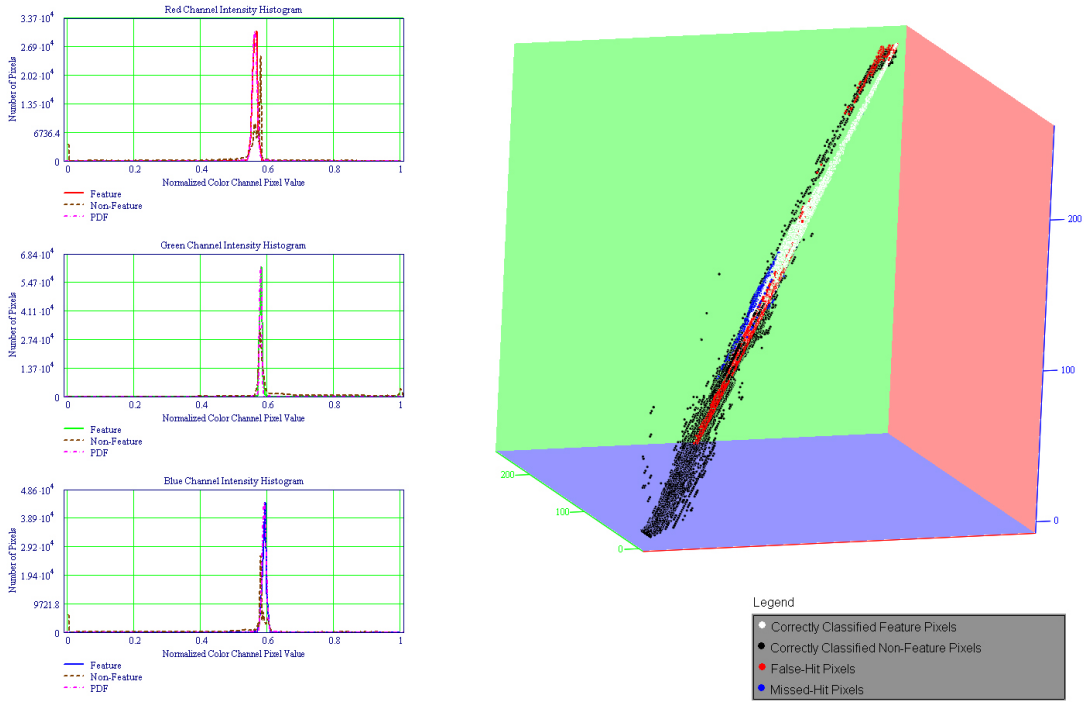

Figure 43: Road Line Normalized Histograms with Univariate Gaussian Overlay (Left). Classifier Results Shown in RGB Space (Right).

Figure 43 also shows the effects of the classifier on the data in RGB space. The results are very poor, although it is not apparent from the graph just how bad they are. There were very few feature pixels missed by the classifier resulting in only 0.4% error. Unfortunately, the false hits were vast resulting in a 59% error. So for this classifier, more than half of the non-feature pixels were selected as feature pixels. This classifier is essentially useless for this test case. Figure 44 shows the road line test image processed with the color direction classifier, with the pixel assumed to be features shown in yellow. Color direction yields even worse results with the road line images than the color range classifier. In this case, almost the entire road has been selected as a road line.

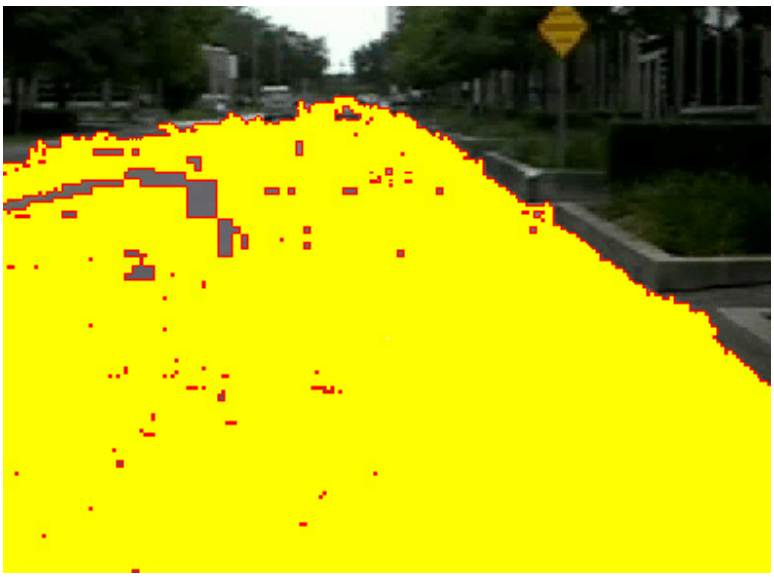

Figure 44: Road Lines Processed with Color Direction Classifier. Classifier Hits are Shown in Yellow.

The color direction classifier essentially is a specialized classifier that should be used on features that are fairly unique in the image and are likely to undergo drastic changes in lighting conditions. This image processing method is fairly quick in relation to some of the more complicated versions and can be used as long as the feature data direction is fairly separated from the ambient data.

3D Color Gaussian

The three-dimensional color Gaussian is the first of the real statistical classification techniques discussed in this paper. A three dimensional Gaussian model is basically an ellipsoid of data in space, see Figure 45. This ellipsoid represents the data contained by the distribution at a given standard deviation. Each value of standard deviation will represent a different ellipsoid having the same center point and ratios, just different in size.

The advantage that the Gaussian model has over the previous methods is that the variance of each of the color channels is correlated and creates a much better overall model for a specific color. The disadvantage of the 3D Gaussian is that there is none of the built in features for handling data inconsistencies, such as changes in lighting. Once a model has been created in training data, the classifier will only find pixels that are fairly close to the trained distribution. In many cases this is a reasonable request and the 3D Gaussian classifier works very well, but as with most aspects of this research, the results are dependent on the situation.

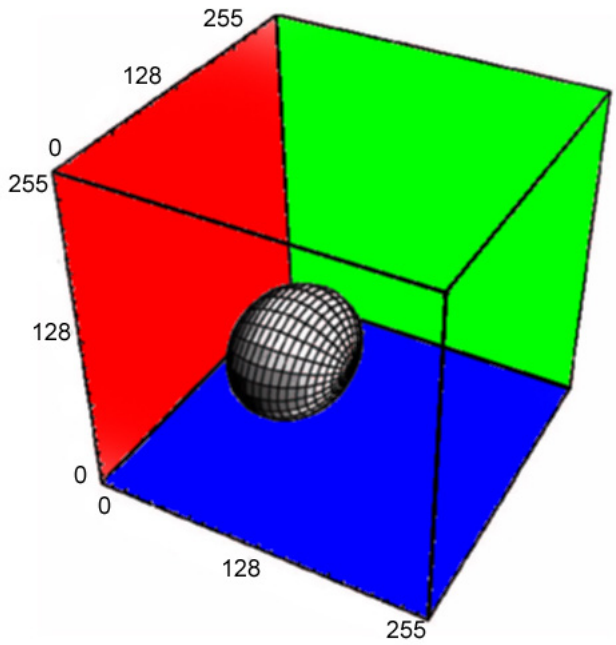

Figure 45: 3D Color Gaussian Distribution Profile in RGB Space

Mathematics

The mathematics for a three-dimensional Gaussian is a bit more complicated than the univariate case that has been discussed already. The mean value is replaced by a

mean vector and the standard deviation is replaced by a covariance matrix, but other than the conversion to multidimensionality, the method is very similar.

Multivariate Gaussian

The general form of the multivariate Gaussian equation is:

$$p(\bar{x}) = \frac{1}{(2\pi)^{d/2}\left|\overline{\overline{\Sigma}}\right|^{1/2}} \exp\left[-\frac{1}{2}(\bar{x}-\bar{\mu})^{t}\overline{\overline{\Sigma}}^{-1}(\bar{x}-\bar{\mu})\right] \tag{43}$$

Where $p(\bar{x})$ is the PDF of the vector $\bar{x}$ across a multidimensional Gaussian distribution. The quantity d represents the number of dimensions that the Gaussian contains. The mean value has been replaced by the mean vector $\bar{\mu}$ that contains the mean values for every dimension. The standard deviation and variance has been replaced by the $d \times d$ covariance matrix $\overline{\overline{\Sigma}}$ that represents the variance in each dimension, as well as the covariance between dimensions.

For the 3D color Gaussian, the dimensionality of the distribution will obviously be three. The mean vector represents the average values of the red green and blue color channels:

$$\bar{\mu} = \begin{bmatrix} \mu_r \\ \mu_g \\ \mu_b \end{bmatrix} \tag{44}$$

The covariance matrix is a 3×3 matrix representing the variance within and between each color channel. The covariance matrix is defined as:

$$\overline{\overline{\Sigma}} = \begin{bmatrix} \sigma_{rr}^2 & \sigma_{rg}^2 & \sigma_{rb}^2 \\ \sigma_{gr}^2 & \sigma_{gg}^2 & \sigma_{gb}^2 \\ \sigma_{br}^2 & \sigma_{bg}^2 & \sigma_{bb}^2 \end{bmatrix} \tag{45}$$

Since the variances in the covariance matrix represent the product of two color variances, the order of the colors is irrelevant. Then using the commutative property of mathematics, it can be seen that $\sigma_{rg}^2 = \sigma_{gr}^2$, $\sigma_{rb}^2 = \sigma_{br}^2$, and $\sigma_{gb}^2 = \sigma_{bg}^2$. Therefore, the covariance matrix is always symmetric [Dud01].

The values of the mean vector can be calculated the same way in which a non-vector mean would:

$$\begin{bmatrix} \mu_r \\ \mu_g \\ \mu_b \end{bmatrix} = \frac{1}{N} \sum_{i=1}^{N} \begin{bmatrix} red_i \\ grn_i \\ blu_i \end{bmatrix} \tag{46}$$

Where red_i, grn_i, and blu_i represent the color values of the i^{th} pixel of a data set with N points. The variance values of the covariance matrix are also calculated in the same fashion as single variance values:

$$\begin{aligned} \sigma_{rr}^2 &= \frac{1}{N} \sum_{i=1}^{N} (red_i - \mu_r)(red_i - \mu_r) \\ \sigma_{rg}^2 &= \frac{1}{N} \sum_{i=1}^{N} (red_i - \mu_r)(grn_i - \mu_g) \\ &\vdots \\ \sigma_{bb}^2 &= \frac{1}{N} \sum_{i=1}^{N} (blu_i - \mu_b)(blu_i - \mu_b) \end{aligned} \tag{47}$$

Mahalanobis Distance

The exponential part of the multivariate Gaussian equation can be used by itself to determine a data point's amount of membership to the class. The equation:

$$r^2 = (\bar{x} - \bar{\mu})^t \bar{\bar{\Sigma}}^{-1} (\bar{x} - \bar{\mu}) \tag{48}$$

represents the square of the value r which is referred to as the Mahalanobis Distance. The Mahalanobis Distance is the distance that a point, $\bar{x}$, is from the mean of the distribution in standard deviations.

To calculate the Mahalanobis Distance, the inverse of the covariance matrix must first be found. This is a multi-step process that can be done on any square matrix, but will be shown optimized for the 3×3. First, the determinant must be taken of the covariance matrix. For a 3×3, this can be done by taking the sums of the products of the matrix terms along its diagonals:

$$\left|\overline{\overline{\Sigma}}\right| = \begin{vmatrix} \sigma_{rr}^2 & \sigma_{rg}^2 & \sigma_{rb}^2 \\ \sigma_{rg}^2 & \sigma_{gg}^2 & \sigma_{gb}^2 \\ \sigma_{rb}^2 & \sigma_{gb}^2 & \sigma_{bb}^2 \end{vmatrix} = \begin{matrix} \sigma_{rr}^2\sigma_{gg}^2\sigma_{bb}^2 + \sigma_{rb}^2\sigma_{rg}^2\sigma_{gb}^2 + \sigma_{rg}^2\sigma_{gb}^2\sigma_{rb}^2 \\ -\sigma_{rb}^2\sigma_{gg}^2\sigma_{rb}^2 - \sigma_{rr}^2\sigma_{gb}^2\sigma_{gb}^2 - \sigma_{rg}^2\sigma_{rg}^2\sigma_{bb}^2 \end{matrix} \tag{49}$$

Next the adjoint of the covariance matrix must be found. This is done by decomposing the covariance matrix into 2×2 matrices at each element and finding the determinant of the smaller matrix. The negative of the determinant is needed at every other adjoint value so the determinant must be multiplied by $(-1)^{i+j}$. For example, the adjoint's value at its 0,0 element is:

$$adj(0,0) = (-1)^{i+j}\begin{bmatrix} \otimes & \otimes & \otimes \\ \otimes & \sigma_{gg}^2 & \sigma_{gb}^2 \\ \otimes & \sigma_{gb}^2 & \sigma_{bb}^2 \end{bmatrix} = 1\cdot\left(\sigma_{gg}^2\sigma_{bb}^2 - \sigma_{gb}^2\sigma_{gb}^2\right) \tag{50}$$

This is done for all nine elements of the adjoint matrix. The final step is to divide the adjoint by the determinant of the original matrix:

$$\overline{\overline{\Sigma}}^{-1} = \frac{adj\left(\overline{\overline{\Sigma}}\right)}{\left|\overline{\overline{\Sigma}}\right|} \tag{51}$$

Classifying Pixels

Using the equations listed earlier, a 3D Gaussian distribution can be found for a given feature set. This distribution is completely defined by the mean vector and covariance matrix. Using this information the Mahalanobis distance function can be created:

$$r = \sqrt{(\bar{x} - \bar{\mu})^t \overline{\overline{\Sigma}}^{-1} (\bar{x} - \bar{\mu})} \quad (52)$$

The value of *r* gives the number of standard deviations away from the mean color that the current pixel color is. If the pixel value is exactly that of the mean, the Mahalanobis Distance will be zero. The farther away the pixel color value is from the mean, the larger the distance. Depending on the size and shape of the distribution, the Mahalanobis Distance can range from zero to infinity.

The rationale from the univariate Gaussian case still applies. If there is not any error information present to properly determine the classification boundary, a distance of two or three standard deviations is usually a good start. Any pixels that are determined to have a Mahalanobis Distance of less than this value will be added to the feature class, the rest will be added to the non-feature class.

Test on image set #1

Image set #1 has been used as the problematic model for the previous two classification methods, so it's fortunate that the 3D Gaussian classifier works well for it. The road line data fits almost ideally for the 3D Gaussian. The feature data is clustered in a shape the closely resembles an ellipsoid. An unfortunate condition of the data is that non-feature pixels are actually grouped in with the feature pixels. This means that there

are non road line parts of the image that are the exact same color as the road lines. Color information alone will not ever be able to completely separate the road lines from the ambient pixels in the image, but the 3d Gaussian does the best job of any of these classifiers.

The properties of the Gaussian are completely derived from the training data information. The mean vector and covariance matrix can be gotten using the equations in the mathematics section. The only value that needs to be set once the classifier is defined is the Mahalanobis Distance. The 3D Gaussian classifier only determines the distribution of the feature data, it does not account for the accidental finding of non-feature data. Therefore, the Mahalanobis Distance that is optimal for finding the features in the image may not be the distance that yields the least error. In standard pattern recognition a second Gaussian could be defined to represent the non-feature data and the intersection border of the two Gaussians would then become the feature decision boundary. Anything on the feature side of the boundary would be considered to be a feature, and everything on the other side would not.

For this type of classification though, the non-feature data is rarely going to form a Gaussian distribution. Instead, it will typically appear more like noise than a typical distribution function, with peaks and valleys across the spectrum. Because of this, the error of the classifier must be found manually for many different Mahalanobis Distances to determine the optimal case. To do this, an error graph is created that shows the different types of errors for many different Mahalanobis Distances, see Figure 46. The data in the error graphs shown here were created by the automatic tracking data generation program, which is discussed in detail in Appendix C.

It is up to the user to decide which distance value will give the best results for the intended application. Typically, the overall error needs to be minimized, so the Mahalanobis Distance that should be used will be the global minimum of the yellow curve in Figure 46. Some applications may call for a higher consistency in finding the feature, in which case the distance used would be a little higher than the minimum total error. The opposite is true if the application requires that the false hits on non-feature data be minimized.

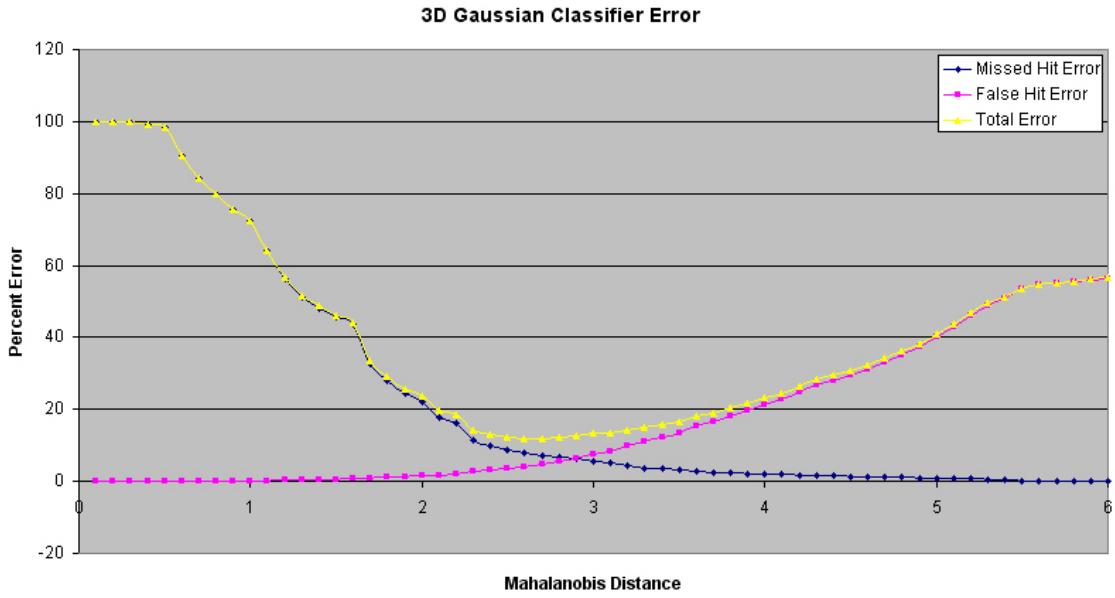

Figure 46: Error Plot for 3D Gaussian Classifier on Road Line Data

For image set #1 a Mahalanobis Distance of 2.7 is used for the distribution that was calculated by the tracking data generation program. The distance of 2.7 gives the overall optimal solution for the 3D Gaussian classifier on this set of data. The errors caused by false hits on non-feature data were at 4.9% which is not too bad. The missed-hit error was a little higher at 7.4%, making the total error 12.3%. This is not great, but this data set is particularly difficult, so 12.3% error is acceptable. The distribution of the classifier can be seen in RGB space in Figure 47.

The road line test image is shown in Figure 48. The pixels classified as road lines are shown in yellow. It is obvious from this image, that the 3D Gaussian distribution works much better than the previous models on detecting road line information. Both road lines visible in this image have been found by the classifier, but some of the road reflections in the distance are still causing problems. These problems are to be expected when processing images with only the color information.

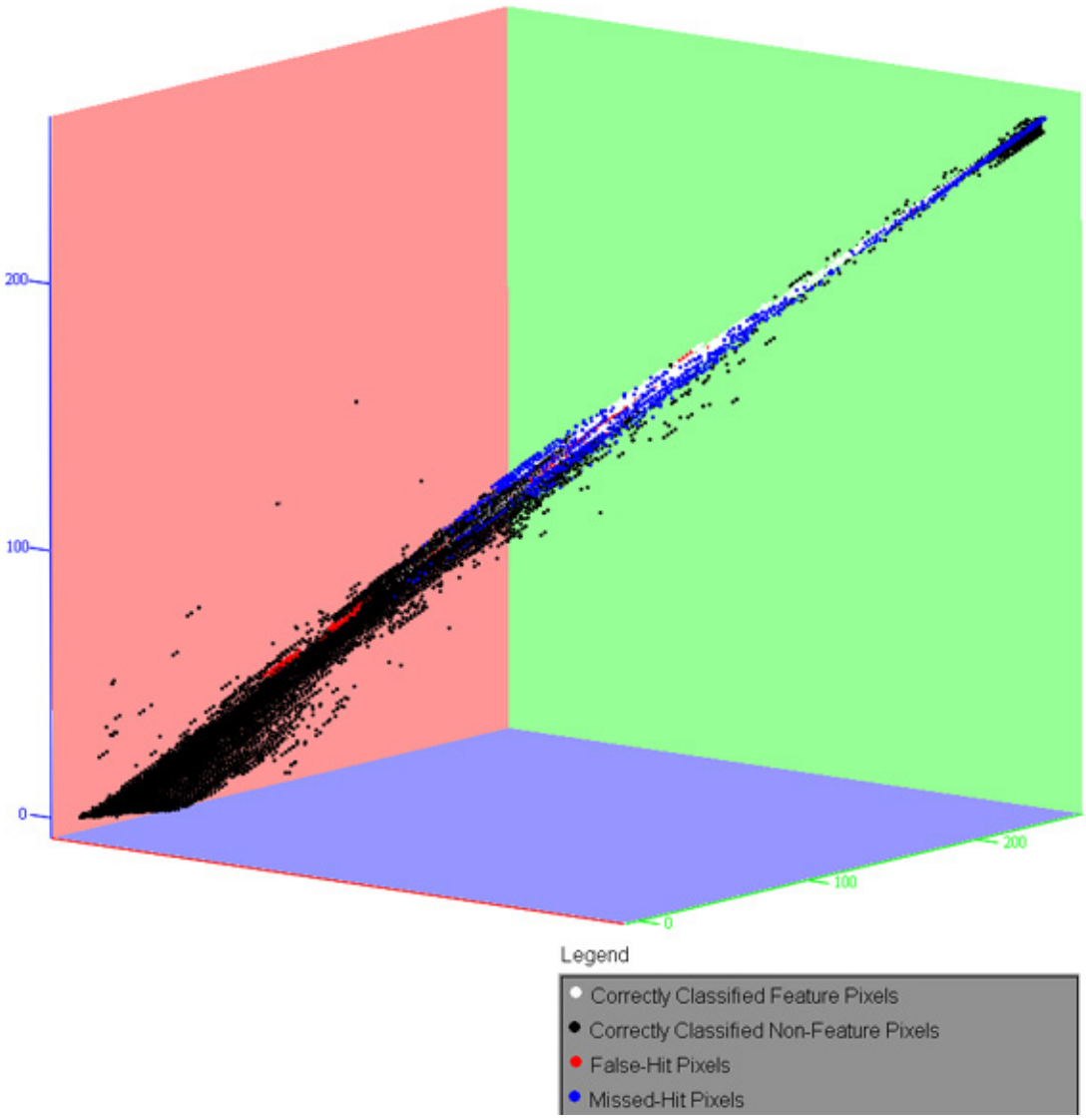

Figure 47: 3D Gaussian Classifier Results Shown in RGB Space.

Figure 48: Road Lines Image Processed with Classifier Hits Shown in Yellow.

Test on image set #2

The second image set is a pretty decent case for a 3D Gaussian classifier. The feature data is well separated from the background and fairly unique in the scene. This works well for the 3D Gaussian model because it is less likely to get false hits.

The error resulting from this classifier is practically insignificant. The false hit errors only occurred at a rate of 0.2% and the missed hits at a rate of 2.3%. This classifier works excellent for the yellow lid images having a total error of only 2.5%. The distribution in RGB space can be seen in Figure 49.

The test image is shown in Figure 50. The results are pretty good for this image. No false hits are present, and only the sides of the lid have escaped detection. These results are surprisingly good, considering the image was taken with different lighting conditions than the training data.

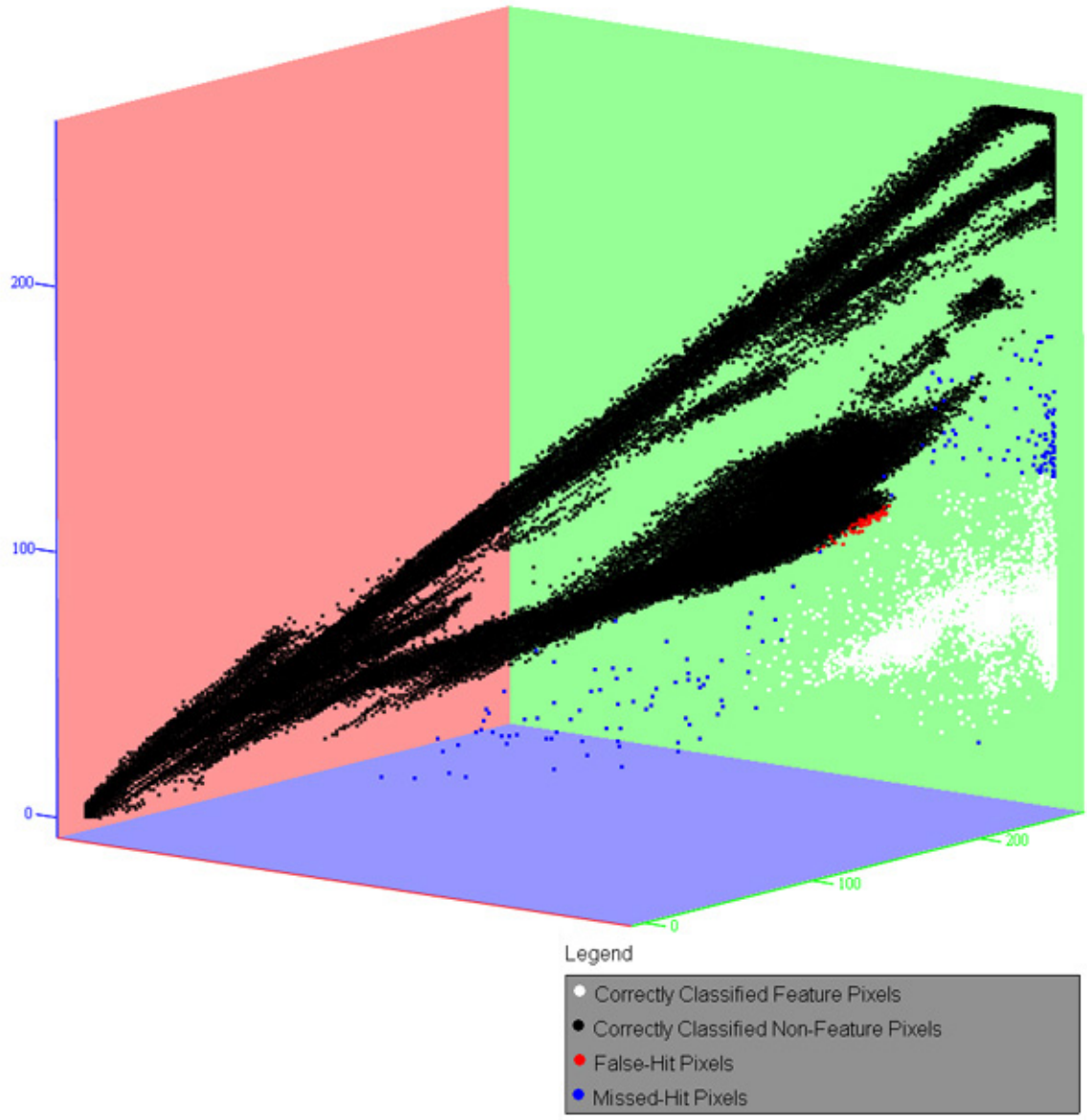

Figure 49: 3D Gaussian Classifier Results Shown in RGB Space.

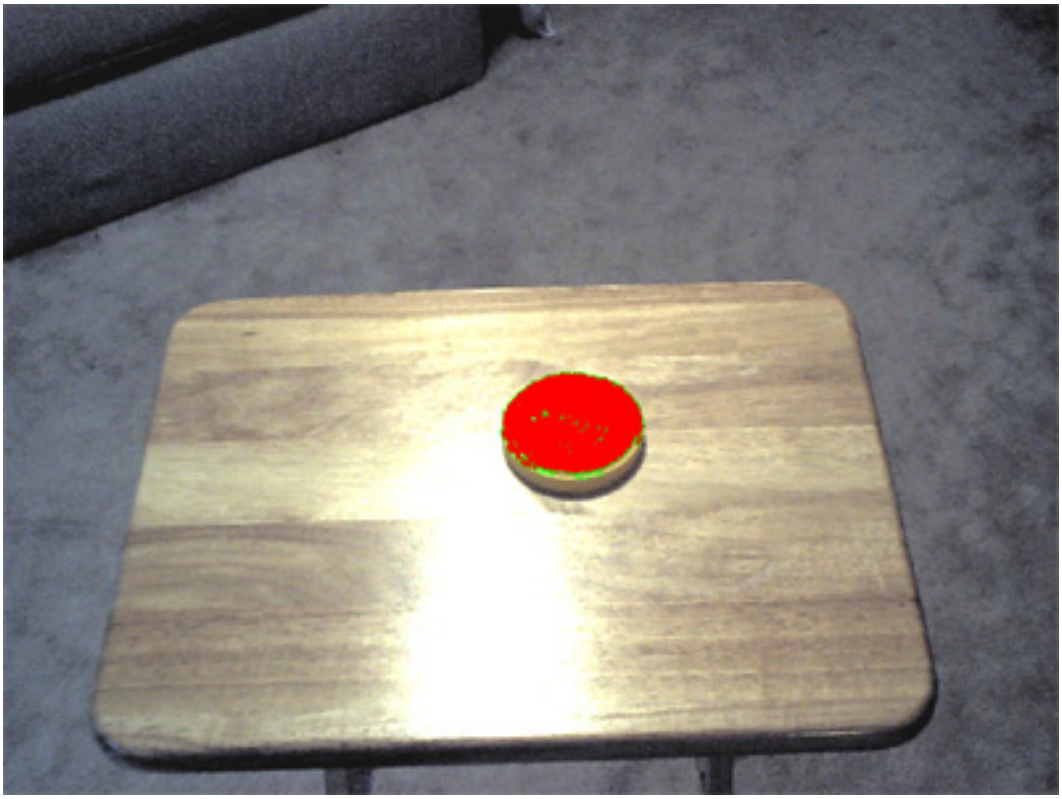

Figure 50: Yellow Lid Image Processed with Classifier Hits Shown in Red.

Test on image set #3

The third image set is a bad case for a Gaussian classifier. The way the data is distributed does not allow for a good distribution to be created. The data is by no means elliptical in shape, not to mention there is a portion of the feature data that overlaps the non-feature data. The false-hit errors were minimal, accounting for only 1.3% error. The missed-hit errors, on the other hand, were much worse at about 29%. The total error between these summed up to be 30.3% error, much too high for good classification.

The results of the classifier are shown in RGB space in Figure 51. The shape of the Gaussian can clearly be seen by observing the ellipse of white points in the graph. The classifier covers very little of the space occupied by the feature data, but this was still the optimal solution for this classifier, because much more error would be incurred if the classifier extended into the data for the non-feature pixels.

The results of this classifier are shown on the warehouse test image in Figure 52. It can be seen that while many of the pixels with the red biohazard logo have been recognized, there are many more that were not. Also, the classifier made a few false hits in places where the color of the barrels was close to the color of the red features in the shadows.

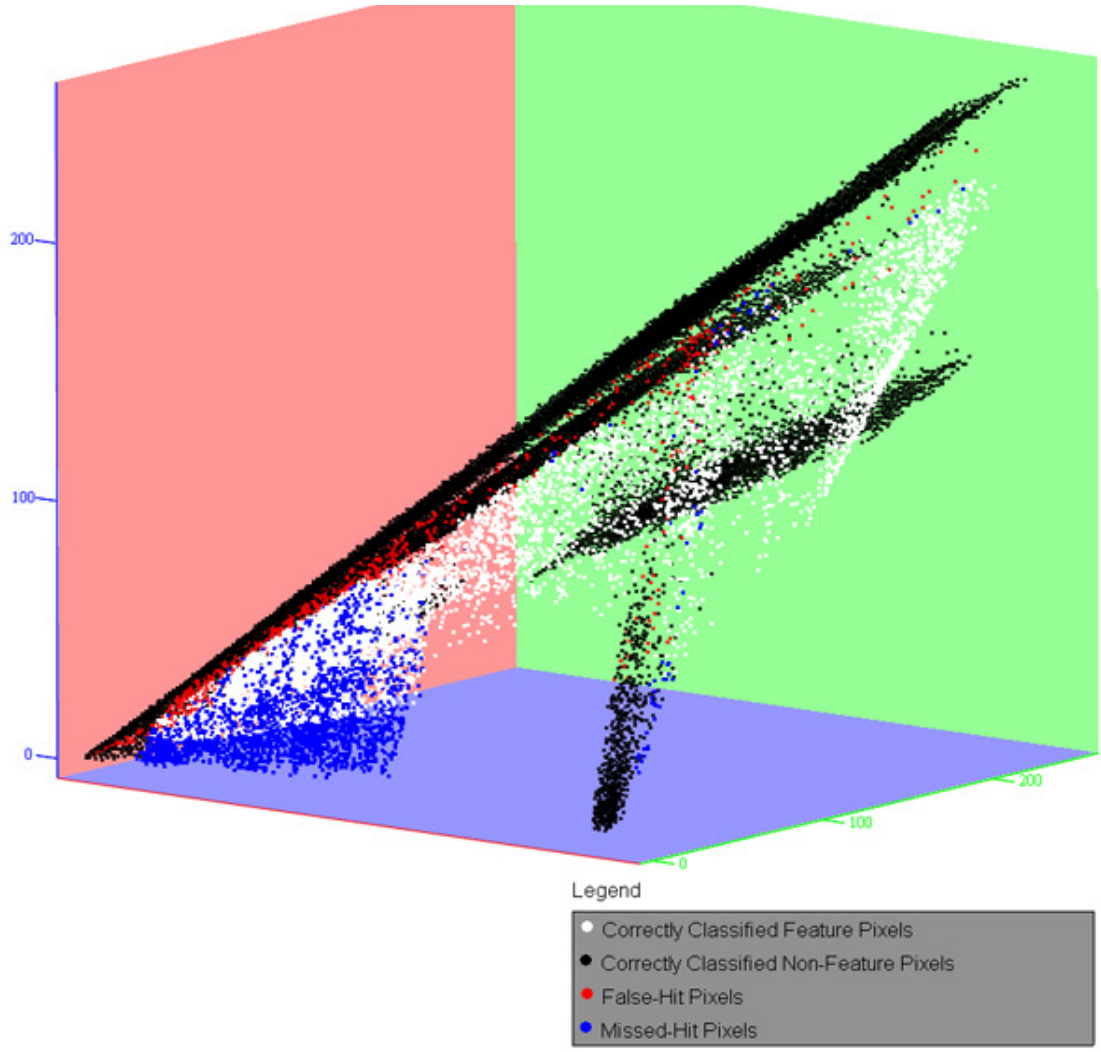

Figure 51: 3D Gaussian Classifier Results Shown in RGB Space.

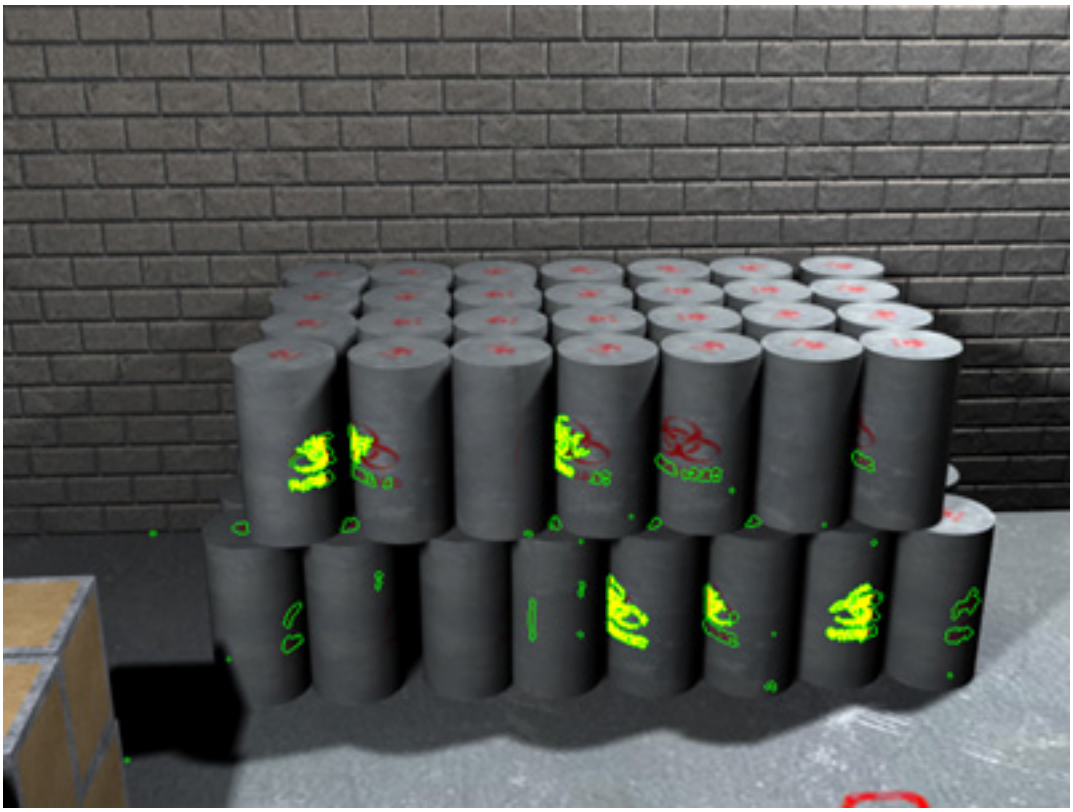

Figure 52: Simulation Warehouse Image Processed with Classifier Hits Shown in Yellow.

2D Normalized Color Gaussian

The concept of the normalized color Gaussian is a hybrid of the 3D Gaussian and the color direction classifiers. This distribution allows for the flexibility of the color range classifier in seeing objects that pass through different lighting conditions. The addition of the Gaussian element allocates a distribution shape that can be modified by a single value, the Mahalanobis Distance.

The procedure in creating this distribution is somewhat different than either of the classifiers that inspired it. Instead of defining a vector that represents the direction of the color in space, the colors themselves are changed from three dimensions to two. This process changes the spatial RGB values to planar XY values that represent the color composition percentage that each pixel has. The resulting color space is no longer a

cube, but a planar triangle, see Figure 53. The process by which this is done will be discussed in the next section.

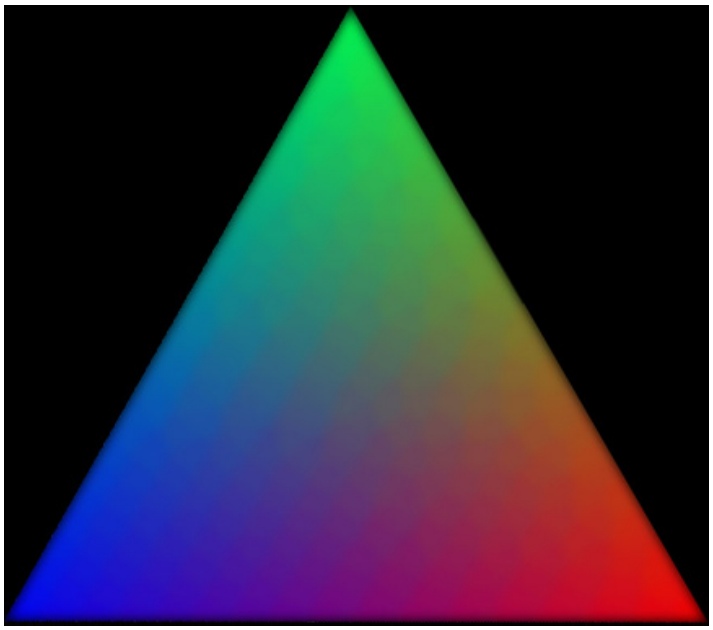

Figure 53: Normalized Color Triangle

Each of the tips of the triangle represents a principle color (red, green, and blue). The mid points along each side of the triangle represent secondary colors (magenta, cyan, yellow). The center of the triangle represents the full spectrum of greys (from white to black). The color intensity information is flattened so that any pixel with the same ratio of component values (red to green to blue) will be located in the same place on the triangle. For instance, the multiples of a shade of cyan: 0,75,75 and 0,150,150 and 0,225,225, are all located at the same point on the normalized color triangle.

Once all of an image's data points are converted to normalized colors, the data set will be a two dimensional distribution that can be dealt with in the same way as the 3D Gaussian. Instead of an ellipsoid in space, the distribution will be defined by an ellipse on a plane. The faster calculations for a 2D Gaussian compliment the extra time needed to convert colors to planar coordinates, leaving a classifier that runs at about the same speed at the 3D Gaussian.

Mathematics

First of all, it should be noted that the conversion of the color information to planar coordinates is not truly normalization. A normalized vector always has a magnitude of one. For this classifier, the procedure is slightly different. The normalization of the color vector comes as a result of decreasing the color dimensionality not from explicitly unitizing the vector.

To convert the RGB coordinates of a pixel to XY normalized color coordinates, a set of three two-dimensional, coplanar direction vectors must be defined. For each color component to have equal weighting on the generation of the new normalized color, the vectors must be equal angles apart from each other. Therefore, the red, green, and blue coordinate transform vectors are defined as:

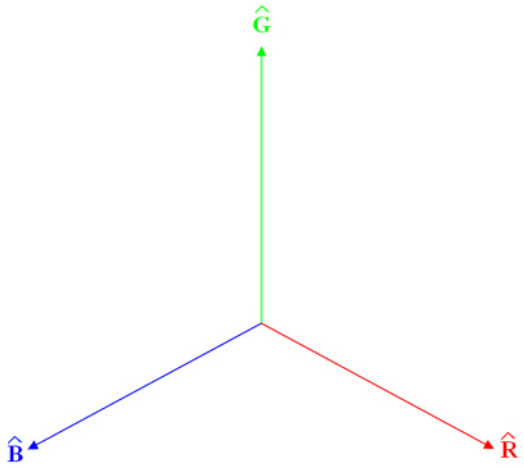

Figure 54: Normalized Color Direction Vectors.

$$\hat{R} = \begin{pmatrix} \frac{\sqrt{3}}{2} \\ -\frac{1}{2} \end{pmatrix}$$

$$\hat{G} = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \tag{53}$$

$$\hat{B} = \begin{pmatrix} -\frac{\sqrt{3}}{2} \\ -\frac{1}{2} \end{pmatrix}$$

Each of these vectors define a direction that is 120° apart from the other two. The three vectors point to the corners of the normalized color triangle, see Figure 54. The vectors are unitized, so the maximum distance that can be traveled in any direction is 1. Therefore, $\hat{R}$, $\hat{G}$, and $\hat{B}$ vectors represent the coordinates of full intensity red, green, and blue in 2D normalized color space, respectively. All possible colors are comprised of these three primary colors , so the 2D normalized color location for any color can be calculated by the equation:

$$\begin{pmatrix} X_i \\ Y_i \end{pmatrix} = \frac{red_i}{red_i + grn_i + blu_i}\hat{R} + \frac{grn_i}{red_i + grn_i + blu_i}\hat{G} + \frac{blu_i}{red_i + grn_i + blu_i}\hat{B} \qquad (54)$$

Where red_i, grn_i, and blu_i, indicate the color channel values for the i^{th} pixel. X_i and Y_i are the 2D normalized color coordinates for the i^{th} pixel.

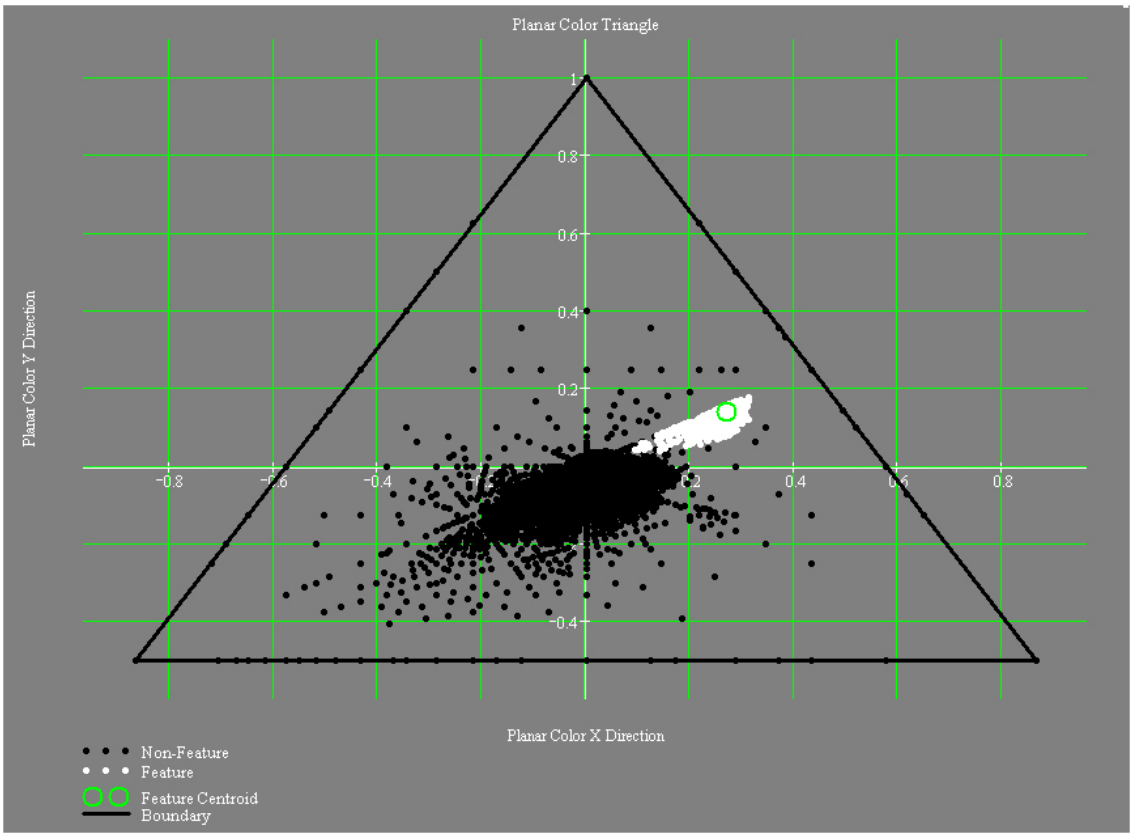

Figure 55: Yellow Lid Image Data Mapped on Normalized Color Plane

This conversion can be done for every pixel in the image and the results can be mapped via an x-y scatter plot, see Figure 55. Now instead of creating a 3D Gaussian ellipsoid to contain relevant data, a less complicated 2D ellipse can be created to represent the data distribution.

The 2D, or bivariate, Gaussian is calculated in the same way as the 3D case. The only difference is the complexity of the calculations is much simpler. The dimensionality of the mean vector drops from three to two:

$$\bar{\mu} = \begin{bmatrix} \mu_x \\ \mu_y \end{bmatrix} \tag{55}$$

The covariance matrix is now a 2x2 matrix representing the normalized color variances:

$$\bar{\bar{\Sigma}} = \begin{bmatrix} \sigma_{xx}^2 & \sigma_{xy}^2 \\ \sigma_{xy}^2 & \sigma_{yy}^2 \end{bmatrix} \tag{56}$$

The Mahalanobis Distance is used for the 2D Gaussian to separate features from non-features in the same way as for the 3D Gaussian.

$$r = \sqrt{(\bar{x} - \bar{\mu})^t \bar{\bar{\Sigma}}^{-1} (\bar{x} - \bar{\mu})} \tag{57}$$

The results obtained from the 2D Gaussian have the same meaning as the 3D as well. The quantity *r* represents the number of standard deviations away point $\bar{x}$ is from the mean. Calculating the $\bar{\bar{\Sigma}}^{-1}$ for the bivariate case is much easier and can be represented by the equation:

$$\bar{\bar{\Sigma}}^{-1} = \frac{1}{\sigma_{xx}^2 \sigma_{yy}^2 - \sigma_{xy}^4} \begin{pmatrix} \sigma_{yy}^2 & -\sigma_{xy}^2 \\ -\sigma_{xy}^2 & \sigma_{xx}^2 \end{pmatrix} \tag{58}$$

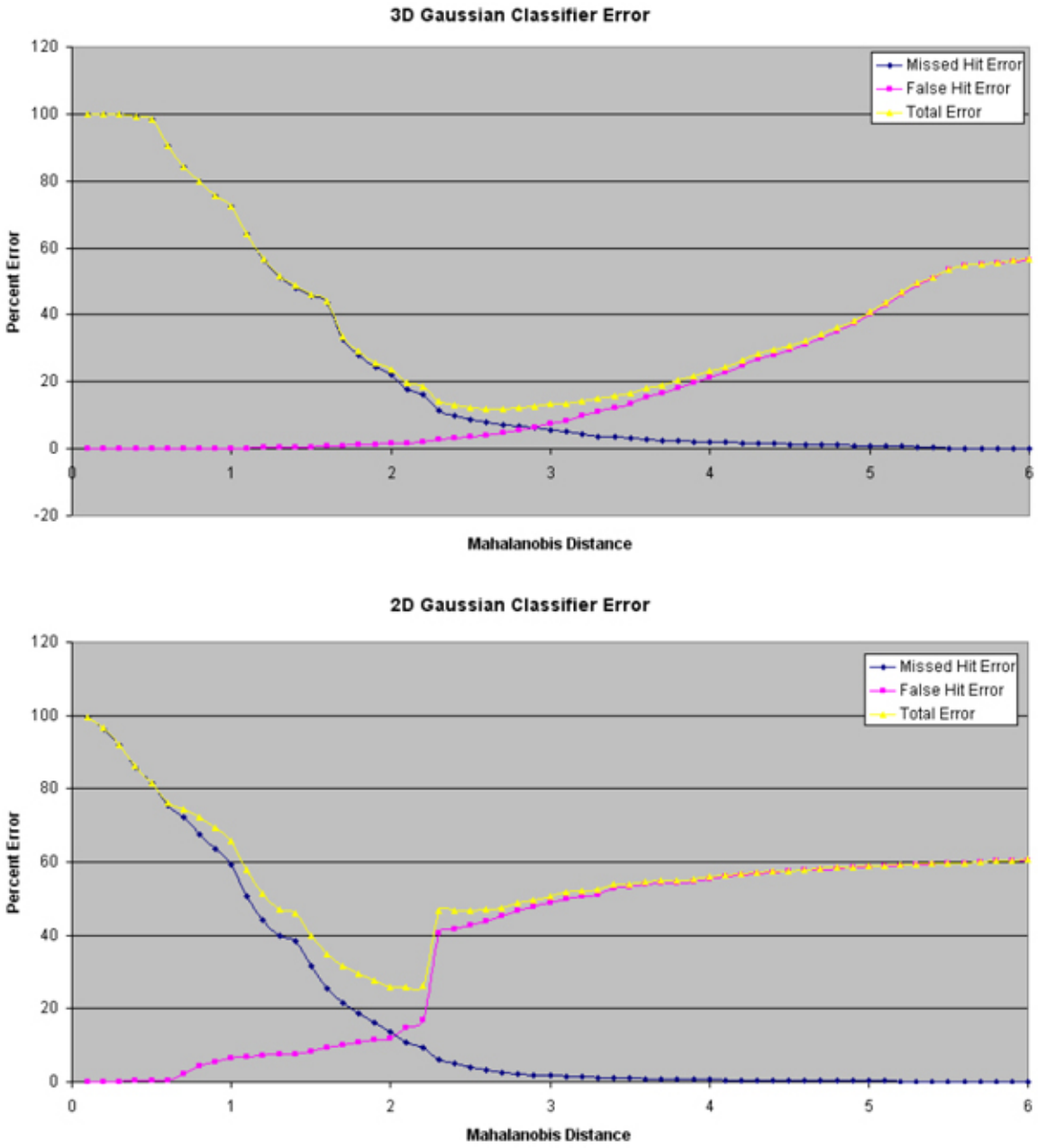

Figure 56: Road Line Data Error Plots. 3D Gaussian Error Plot (Top). 2D Normalized Gaussian Error Plot (Bottom).

When creating the training data for the model, each pixel's information must be converted to normalized coordinates. The mean vector and covariance matrix is calculated for the training set. When this classifier is used on an image, each pixel must be converted to the normalized coordinate system, and the Mahalanobis Distance must be calculated for this normalized point. If the distance is less than the threshold value, the

point is considered to be a feature. Otherwise, the data point is considered to be part of the non-feature set.

Again, this classifier only accounts for the probability of a feature occupying a certain point. To find the optimal classifier, error plots must be made. These plots are identical in nature to the 3D Gaussian error plots, but the error curves have more impulsive trends. Since image information is compressed into a smaller area, the occurrence of errors tends to happen abruptly and increase quickly, see Figure 56.

Test on image set #1

The road line data causes the same problems with the 2D normalized Gaussian classifier as the color range. Since the normalized colors are so close to each other the distinction between the features and non-features is difficult. The classifier results are shown in Figure 57.

The feature class in this data set occupies a very small region of the normalized color space. This area is so small in fact, that it has been magnified to be visible on the graph. The feature data is very condensed at this point, but as with the color direction classifier, the non-feature data is densely packed in this location as well. The results of this classifier are very poor. Missed-hit errors were approximately 14.0% while the false-hit errors were around 12.4%. The total error, therefore, was about 26.4%. This high error rate can be seen in the test image in Figure 58.

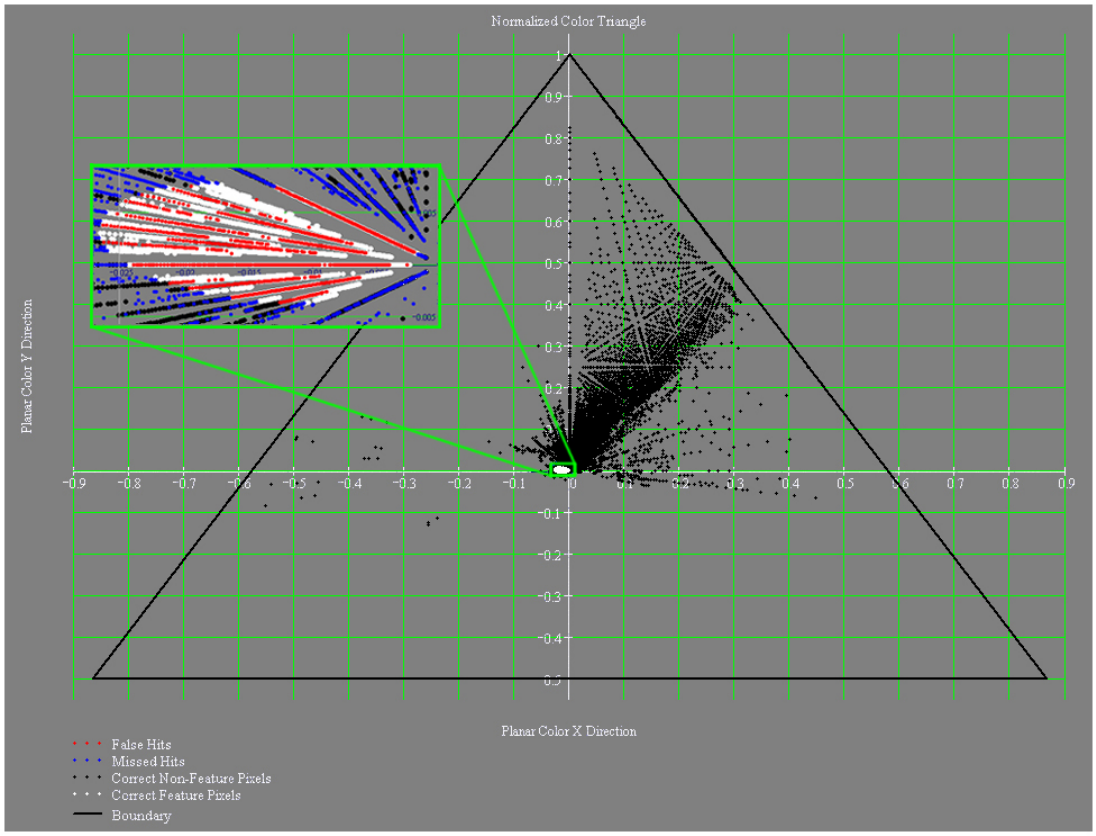

Figure 57: 2D Gaussian Classifier Results Shown in Normalized Color Space.

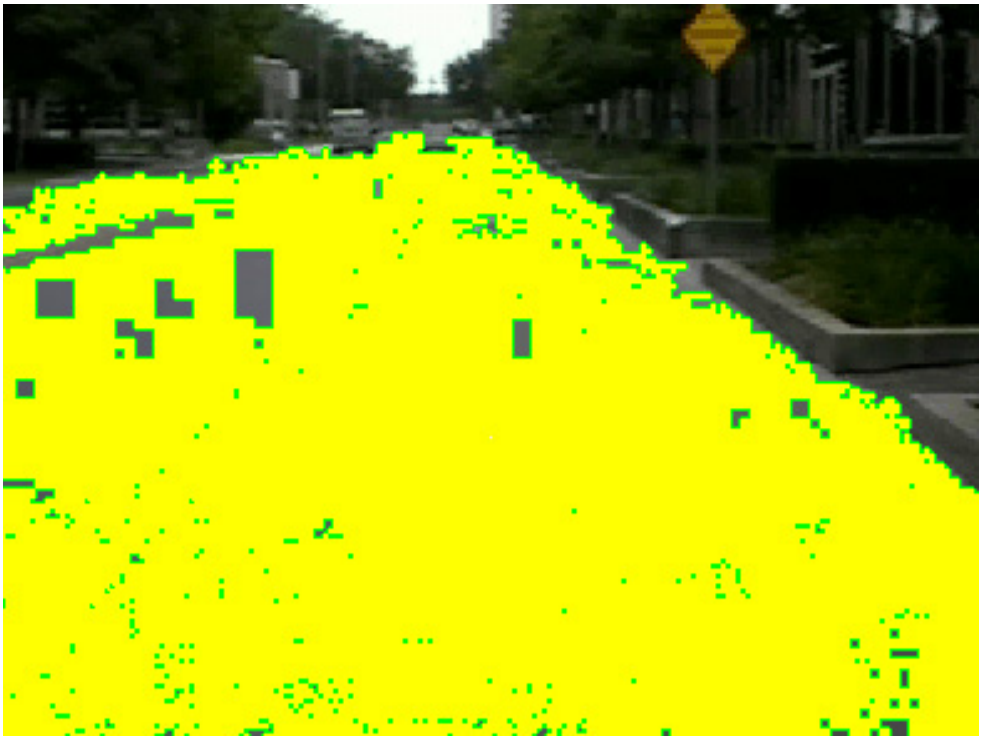

Figure 58: Road Line Image Processed with 2D Gaussian Classifier.

Test on image set #2

The yellow lid data stays consistently easy to classify. The feature data in this set is found without difficulty by the normalized color Gaussian model, much like it has been with subsequent efforts. The data separation of the feature and non-feature regions allow for comfortable classification results, see Figure 59.

The 2D normalized Gaussian classifier gave practically no false hits at less than 0.1% error and very low missed-hit errors at 1.9%. Once again, the yellow-lid data is an excellent feature set for classification. The results of the 2D normalized Gaussian classifier on the test image can be seen in Figure 60.

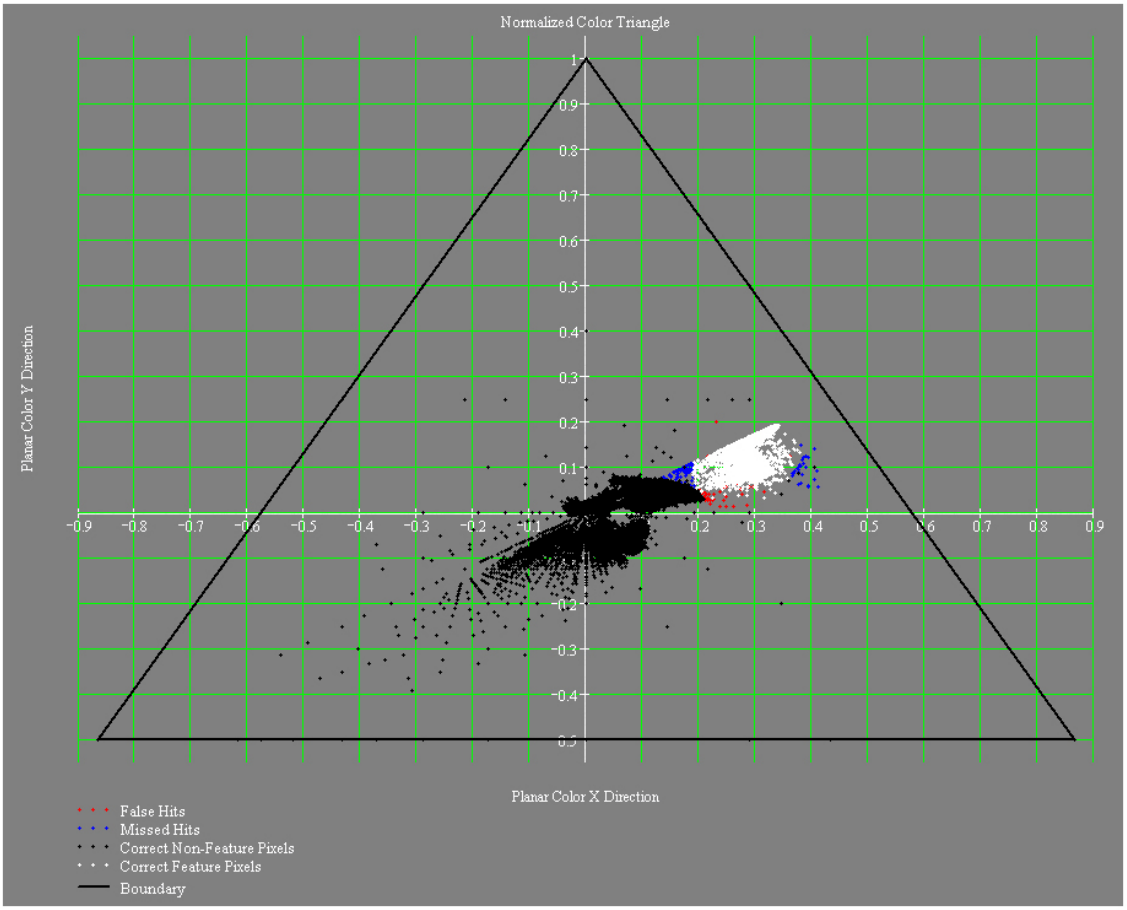

Figure 59: 2D Gaussian Classifier Results Shown in Normalized Color Space.

Figure 60: Yellow Lid Image Processed with 2D Gaussian Classifier.

Test on image set #3

The simulated warehouse images proved to be more difficult for the 2D Gaussian classifier than for the color direction. Although, these methods have similar components, the simulated data still causes confusion with a Gaussian based classifier. When normalized, the feature data falls very close to the non-feature data.

The optimal solution for this classifier was to make the distribution shape smaller to reduce false hits. In turn, the amount of missed hits is very high. The false hits came in at about 1.3% error and the missed hits, about 24.4%. The total error of this classifier is about 25.7%. The image used to test this classifier worked exceptionally well considering the high error rate on the training data. This image happened to have a high amount of reds in the properly classified range, and therefore did a reasonably good job with finding the biohazard logos on the barrels.

It's important to note that this classifier works well on this particular image, but would probably not work so well on other images from this simulation. The training data should represent all of the probable scenarios and build a reliable classifier that accounts for the most frequent of these. This particular data set just happens to be in the most-frequent-case data and does not represent many of the other possibilities.

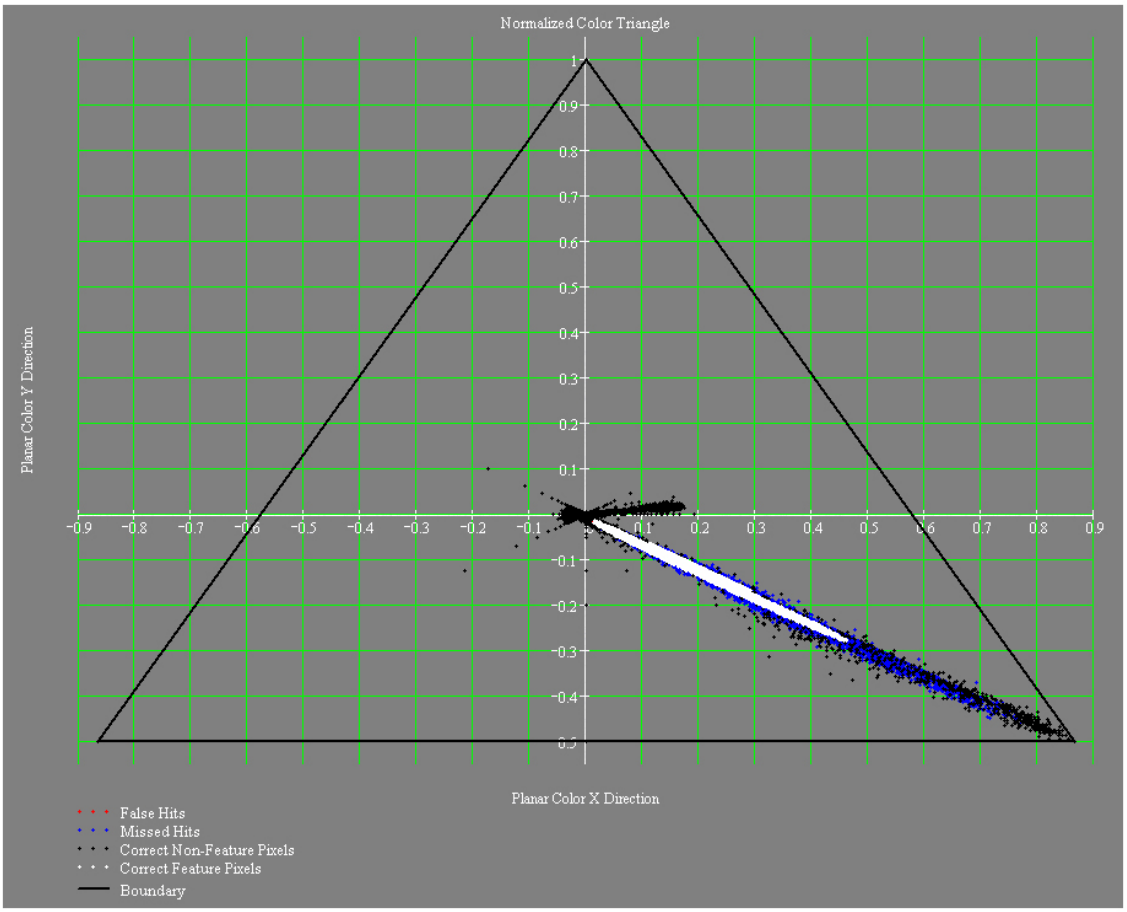

Figure 61: 2D Gaussian Classifier Results Shown in Normalized Color Space.

Classifier Selection

By now it should be apparent that each of the classifiers has strong and weak points. The current situation is always a strong factor in which of these classifiers, if any, are going to work. There are many more classifiers available, outside of the scope of this

document, that can be used if these don't work for a given application. For this research though, these four techniques were deemed sufficient.

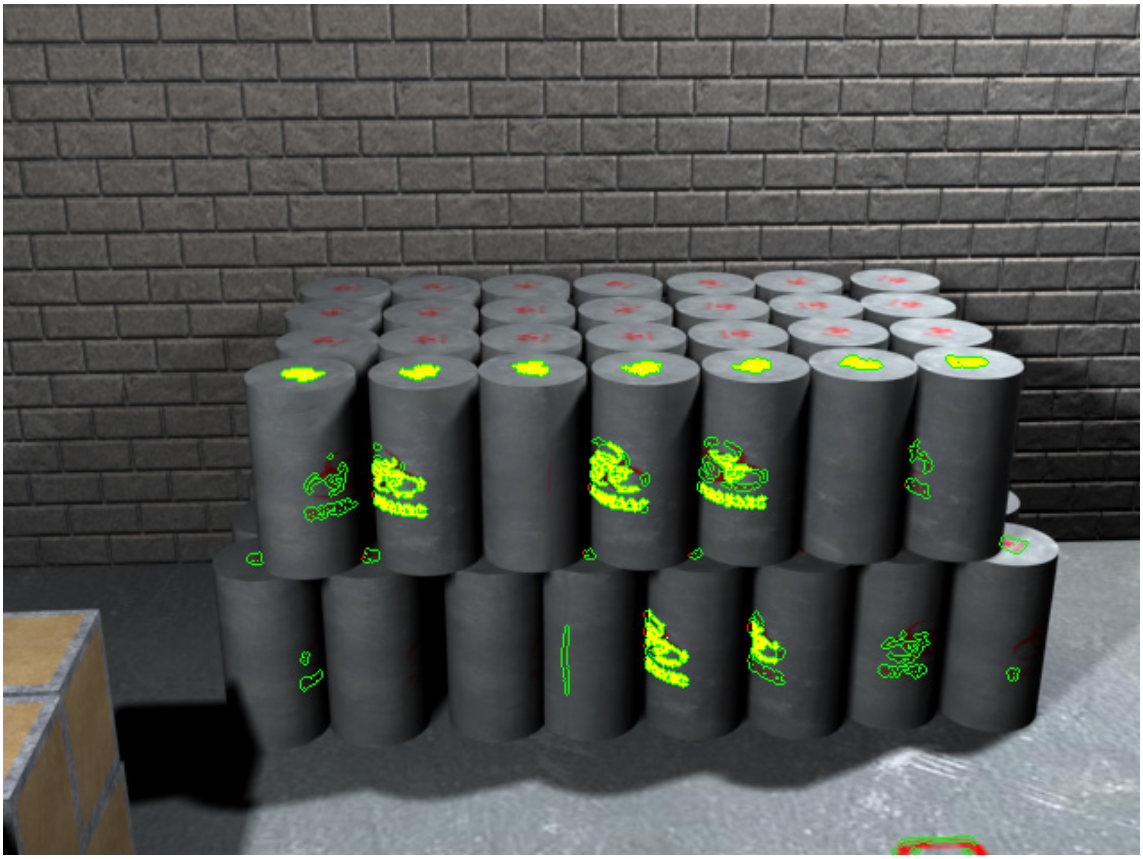

Figure 62: Simulation Warehouse Image Processed with Classifier Shown in Yellow.

Every time a new tracking situation arises, it's important to spend some time considering the conditions and requirements of the operation. The environment is first on the list. Depending on the application, a vehicle may undergo an onslaught of different circumstances. Outdoor vehicles are likely to encounter drastic variances in lighting conditions, higher rates of speed, greater ranges, etc. An indoor vehicle, on the other hand, may be slower and in more consistent lighting, but require very accurate positioning.

Much of the classifier selection relies on the features that are specified for a vehicle. This is an in-depth process that is discussed in a later chapter, but suffice to say

that it is preferable to have features that stand out in the environment and that can be seen from wherever the vehicle might travel. If features are wisely chosen, the less computationally taxing classification methods might be useable, allowing for faster and more accurate tracking.

CHAPTER 4
PLANAR VISUAL POSITIONING CONCEPT

When a three-dimensional scene is converted to a two-dimensional image some information is lost, namely the perception of depth. From a single image alone, it is not possible to reconstruct a full three-dimensional scene. Visual scene reconstruction is typically done with the aid of a non-visual sensor (i.e. laser range finder, sonar, radar, etc), or through stereovision. Stereovision allows three-dimensional reconstruction by checking for disparity between images from two offset cameras of known positions. This information can be processed and used to determine feature depth in the image.

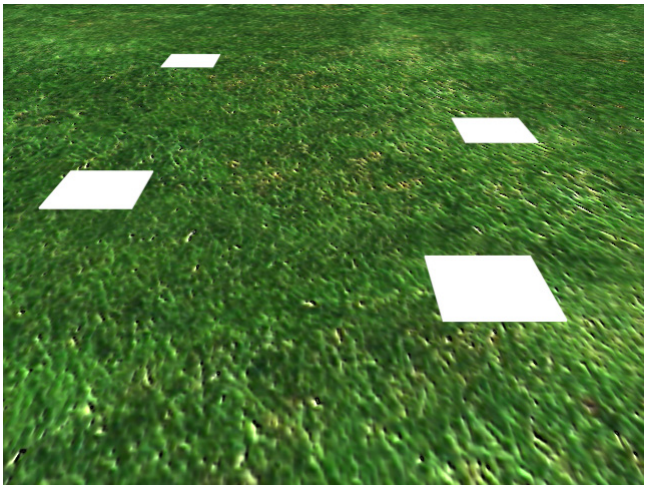
Figure 63: Ground Feature Simulation

Stereovision triangulates the three-dimensional placement of every object in the scene, and is therefore inherently complex. In some instances, specifically in planar visual positioning, not all of this information is necessary. In planar positioning the object that is being tracked resides on a known plane, typically the ground. This plane's position acts as a constraint that allows for depth perception from a single image. For example, Figure 63 shows a simulation image that is used for camera calibration. This image illustrates a grassy surface with a series of white squares painted on it. In this

image everything that is seen is contained in the ground plane. A stereovision system would probably be able to determine the x,y planar coordinates of the painted squares, but calculating information from two images is unwarranted when everything that is needed can be found from one. In this image, knowing the camera information and the ground plane coordinates is sufficient to fill in all of the unknowns and completely portray this image in three-dimensional space from a single two-dimensional image.

Determining planar distances from a two-dimensional image is fairly simple conceptually. The camera properties, intrinsic and extrinsic, along with a few other variables must be known prior to calculating any distances. The extrinsic camera properties are those which define the camera's physical location and orientation according to a pre-defined world coordinate system (WCS). Extrinsic camera properties are displayed in Table 3.

Table 3: Extrinsic Camera Properties

Variable	Description
x	Camera's 'x' coordinate
y	Camera's 'y' coordinate
z	Camrea's 'z' coordinate
θ_p	Camera's pan angle (rotation about the z-axis)
θ_t	Camera's tilt angle (rotation about the x-axis)
θ_s	Camera's slant angle (rotation about the y-axis)

The intrinsic properties convey the inner workings of the camera, namely the field-of-view (FOV). Table 4 shows the intrinsic camera properties.

Table 4: Intrinsic Camera Properties

Variable	Description
ψ_h	Camera FOV in the horizontal
ψ_v	Camera FOV in the vertical

The final information needed comes from the video capture device and the equation of the ground plane.

Table 5: Other Properties

Variable	Description
N_h	Number of captured pixels in horizontal (Horizontal resolution of image)
N_v	Number of captured pixels in vertical (Vertical resolution of image)
A	'x' component of plane's normal vector
B	'y' component of plane's normal vector
C	'z' component of plane's normal vector
D	Plane's offset from the WCS origin

The planar equation [D;A,B,C] is the projective geometry representation of a plane in space, where the vector (A,B,C) indicates a vector normal to the plane. The scalar D indicates the normal distance of the plane from the origin. This concept will be discussed further in the next section.

The mathematics required to spatial coordinates are three-dimensional in nature, but can compensate for any position and orientation of the camera. Using Euclidean mathematics for these calculations can become extremely complex and tedious. For this reason projective geometry was used instead.

Projective Geometry

Projective geometry, while less frequently used than Euclidean mathematics is superior in many ways. For instance, projective geometry has a built in mechanism to account for points at infinity and tends to be much more powerful for three-dimensional calculations than Euclidean mathematics. Also, coordinates and functions represented in projective geometry lend themselves to matrix operations well, vastly improving calculation time for complex equations. These properties of projective geometry make it the key to determining the three-dimensional range points.

Before the process of calculating three-dimensional distance can be explained, the reader must have a basic understanding of projective geometry. The fundamental concepts of points, planes, and lines will be discussed in the following sections as an introduction to the more complex relationships [Cra03].

Equation of a Point

The vector $\overline{r_1}$ to a point Q_1 in space from the reference point O can be expressed as:

$$\overline{r_1} = \frac{x_1\hat{i} + y_1\hat{j} + z_1\hat{k}}{w_1} \tag{59}$$

or

$$\overline{r_1} w_1 = \overline{S_{01}} \tag{60}$$

where $\overline{S_{01}} = x_1\hat{i} + y_1\hat{j} + z_1\hat{k}$ and $w_1=1$ and (x_1,y_1,z_1) are the normal Cartesian coordinates of the point Q_1. The subscript '0' represents that S_{01} is dependent on the

selection of the origin. The coordinates of a general point Q can be expressed in the same way as:

$$\bar{r}w = \overline{S_0} \tag{61}$$

where $\overline{S_0} = x\hat{i} + y\hat{j} + z\hat{k}$. The purpose of having the 'w' present is to make it possible to display coordinates at infinity. As w approaches 0, points x, y, and z approach ∞. When w = 0, point Q is at a point at ∞ parallel to the direction of $\overline{S_0}$.

The absolute distance of the point Q from the reference point O can be shown by:

$$\left|\bar{r}\right| = \frac{\left|\overline{S_0}\right|}{|w|} \tag{62}$$

where when w = 1 the distance of Q from the origin is equal to $\sqrt{x^2 + y^2 + z^2}$ and when w = 0 it is infinite. The equation of a point in projective geometry is displayed as:

$$(w; x, y, z) \tag{63}$$

The 'w' is separated from x, y, and z by a semicolon to indicate there is a difference in the units.

Equation of a Plane

The equation of a plane is analogous to the equation of a point, in fact they are considered dual. The equation of a plane that passes through a point Q_1 and is normal to a vector $\overline{S} = A\hat{i} + B\hat{j} + C\hat{k}$ can be expressed as:

$$(\bar{r} - \overline{r_1}) \cdot \overline{S} = 0 \tag{64}$$

where $\bar{r} = x\hat{i} + y\hat{j} + z\hat{k}$ is a vector from the origin to any point on the plane. Equation (64) can be rewritten as:

$$\bar{r} \cdot \bar{S} + D_0 = Ax + By + Cz + D_0 = 0 \tag{65}$$

where:

$$D_0 = -\bar{r_1} \cdot \bar{S} = -(Ax_1 + By_1 + Cz_1) \tag{66}$$

…so a plane can be represented in the form:

$$[D_0;A,B,C] \tag{67}$$

where, again, the semicolon is used to separate the values by their dimension types.

Equation of a Line

A line can be defined in projective geometry by two points. These points $\bar{r_1}(x_1, y_1, z_1)$ and $\bar{r_2}(x_2, y_2, z_2)$ define a vector $\bar{S}$ which can be shown by:

$$\bar{S} = (\bar{r_2} - \bar{r_1}) \tag{68}$$

This equation for $\bar{S}$ can alternatively be expressed as:

$$\bar{S} = L\hat{i} + M\hat{j} + N\hat{k} \tag{69}$$

where $L = x_2 - x_1$, $M = y_2 - y_1$, and $N = z_2 - z_1$. These values are defined as the direction ratios of the line. Squaring Equation (69) yields:

$$L^2 + M^2 + N^2 = |S|^2 \tag{70}$$

Solving this equation for L, M, and N gives the unit direction ratios (or direction cosines) of the line.

$$L = \frac{x_2 - x_1}{|S|}$$

$$M = \frac{y_2 - y_1}{|S|} \tag{71}$$

$$N = \frac{z_2 - z_1}{|S|}$$

For this case Equation (70) reduces to:

$$L^2 + M^2 + N^2 = 1 \tag{72}$$

Let $\bar{r}$ designate a vector from the origin to any other point on the line. The vector $\bar{r} - \bar{r_1}$ will clearly be parallel the vector $\bar{S}$. Therefore, we can write:

$$(\bar{r} - \bar{r_1}) \times \bar{S} = \bar{0} \tag{73}$$

Defining S_0 as:

$$\bar{r_1} \times \bar{S} = \bar{S_0} \tag{74}$$

Gives the moment of the line about the origin. Rearranging equation (74) yields:

$$\bar{r} \times \bar{S} = \bar{S_0} \tag{75}$$

From this equation it is apparent that $\bar{S}$ and $\bar{S}_0$ are perpendicular, so they must satisfy the orthogonality condition:

$$\bar{S} \cdot \bar{S_0} = 0 \tag{76}$$

Expanding Equation (74) gives:

$$\overline{S_0} = \begin{bmatrix} \hat{i} & \hat{j} & \hat{k} \\ x_1 & y_1 & z_1 \\ L & M & N \end{bmatrix} \tag{77}$$

which can be expressed as:

$$\overline{S_0} = P\hat{i} + Q\hat{j} + R\hat{k} \tag{78}$$

where:

$$\begin{aligned} P &= y_1 N - z_1 M \\ Q &= z_1 L - x_1 N \\ R &= x_1 M - y_1 L \end{aligned} \tag{79}$$

Equation (76) is expanded to give:

$$LP + MQ + NR = 0 \tag{80}$$

The values of L, M, N, P, Q, and R define a line. This notation is referred to as the Plücker Coordinates of a line, and is typically written as {L,M,N;P,Q,R} or $\{\overline{S}, \overline{S_0}\}$.

A Line and a Plane Define a Point

The coordinates of the line and plane are given as $\{\overline{S_1}, \overline{S_{01}}\}$ and $[D_2, \overline{S_2}]$, respectively. Their equations may be rewritten as:

$$\overline{r_1} \times \overline{S_1} = \overline{S_{01}} \tag{81}$$

and

$$\overline{r_2} \cdot \overline{S_2} + D_2 = 0 \tag{82}$$

where $\overline{r_1}$ and $\overline{r_2}$ are vectors to any point on the line and plane, respectively. At the intersection point (indicated by the vector $\bar{r}$), conditions for both Equation (81) and (82) must be met so:

$$\bar{r} \times \overline{S_1} = \overline{S_{01}} \tag{83}$$

and

$$\bar{r} \cdot \overline{S_2} + D_2 = 0 \tag{84}$$

Forming a vector product of $\overline{S_2}$ with Equation (83) gives:

$$\overline{S_2} \times \left(\bar{r} \times \overline{S_1}\right) = \overline{S_2} \times \overline{S_{01}} \tag{85}$$

Expanding the left side of Equation (85) gives:

$$\bar{r}\left(\overline{S_2} \cdot \overline{S_1}\right) - \overline{S_1}\left(\overline{S_2} \cdot \bar{r}\right) = \overline{S_2} \times \overline{S_{01}} \tag{86}$$

Substituting Equation (82) gives:

$$\bar{r}\left(\overline{S_2} \cdot \overline{S_1}\right) = \overline{S_2} \times \overline{S_{01}} - D_2 \overline{S_1} \tag{87}$$

Finally, the point of intersection is:

$$\left[\overline{S_2} \cdot \overline{S_1}; \overline{S_2} \times \overline{S_{01}} - D_2 \overline{S_1}\right] \tag{88}$$

Coordinate System Translation and Rotation Matrices

An object in space has six degrees of freedom; translation in the x, y, and z directions and rotation about the x, y, and z axes. To completely modify an object's location in the world coordinate system (WCS), all translational and rotational components must be available for modification. This concept is essential for planar

positioning. This will allow the camera to be placed at any position and orientation in space.

First the coordinate system must be moved to its new location using the translation matrix $\overline{\overline{R_d}}$, see Equation (89). The values dx, dy, and dz are the linear displacement values that depict the offset in the x, y, and z (respectively) from the current location in the WCS:

$$\overline{\overline{R_d}} = \begin{bmatrix} 1 & 0 & 0 & dx \\ 0 & 1 & 0 & dy \\ 0 & 0 & 1 & dz \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{89}$$

$\overline{\overline{R_d}}$ is the simplest of the translation and rotation matrices, yet it accounts for three degrees of freedom in the coordinate system.

Next, the coordinate system must be rotated to it's new orientation. $\overline{\overline{R_t}}$ is the rotation matrix that accounts for camera tilt (rotation about the x-axis). All angles are measured in the right hand sense. Therefore, tilt is measured positive upward starting from the negative z direction.

$$\overline{\overline{R_t}} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\theta_t) & -\sin(\theta_t) & 0 \\ 0 & \sin(\theta_t) & \cos(\theta_t) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{90}$$

$\overline{\overline{R_s}}$ is the rotation matrix that accounts for camera slant (rotation about the y-axis). Slant is measured positive clockwise:

$$\overline{\overline{R_s}} = \begin{bmatrix} \cos(\theta_s) & 0 & \sin(\theta_s) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(\theta_s) & 0 & \cos(\theta_s) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (91)$$

$\overline{\overline{R_p}}$ is the rotation matrix that accounts for camera pan (rotation about the z-axis). Pan is measured positive left:

$$\overline{\overline{R_p}} = \begin{bmatrix} \cos(\theta_p) & -\sin(\theta_p) & 0 & 0 \\ \sin(\theta_p) & \cos(\theta_p) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (92)$$

The overall transformation matrix can be shown as:

$$\overline{\overline{R}} = \overline{\overline{R_d}} \cdot \overline{\overline{R_t}} \cdot \overline{\overline{R_s}} \cdot \overline{\overline{R_p}} \quad (93)$$

Where $\overline{\overline{R}}$ indicates the entire rotation and translation of the coordinate system to the new location specified by dx, dy, dz, θ_p, θ_t, and θ_s. $\overline{\overline{R}}$ is rewritten in terms of these variables in Equation (94). To save space, c(θ) and s(θ) are used to indicate cosine(θ) and sine(θ), respectively.

$$\overline{\overline{R}} = \begin{bmatrix} c(\theta_t)\cdot c(\theta_p) & -c(\theta_t)\cdot s(\theta_p) & s(\theta_t) & dx \\ s(\theta_s)\cdot s(\theta_t)\cdot c(\theta_p)+c(\theta_s)\cdot s(\theta_p) & -s(\theta_s)\cdot s(\theta_t)\cdot s(\theta_p)+c(\theta_s)\cdot c(\theta_p) & -s(\theta_s)\cdot c(\theta_t) & dy \\ -c(\theta_s)\cdot s(\theta_t)\cdot c(\theta_p)+s(\theta_s)\cdot s(\theta_p) & c(\theta_s)\cdot s(\theta_t)\cdot s(\theta_p)+s(\theta_s)\cdot c(\theta_p) & c(\theta_s)\cdot c(\theta_t) & dz \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (94)$$

Creating Range Data

When a camera looks at an image or scene, all the objects that are captured reside in what is known as the viewing volume. The viewing volume is a

pyramid-like portion of three-dimensional space, in which everything is seen by the camera. Inside of this volume, every object is projected on to the image plane along a vector from the object's location in space to the focal point of the camera, see Figure 64. This figure illustrates the viewing volume and image plane projection for the simulation image at the beginning of this chapter. It is apparent how the image is created from the original objects.

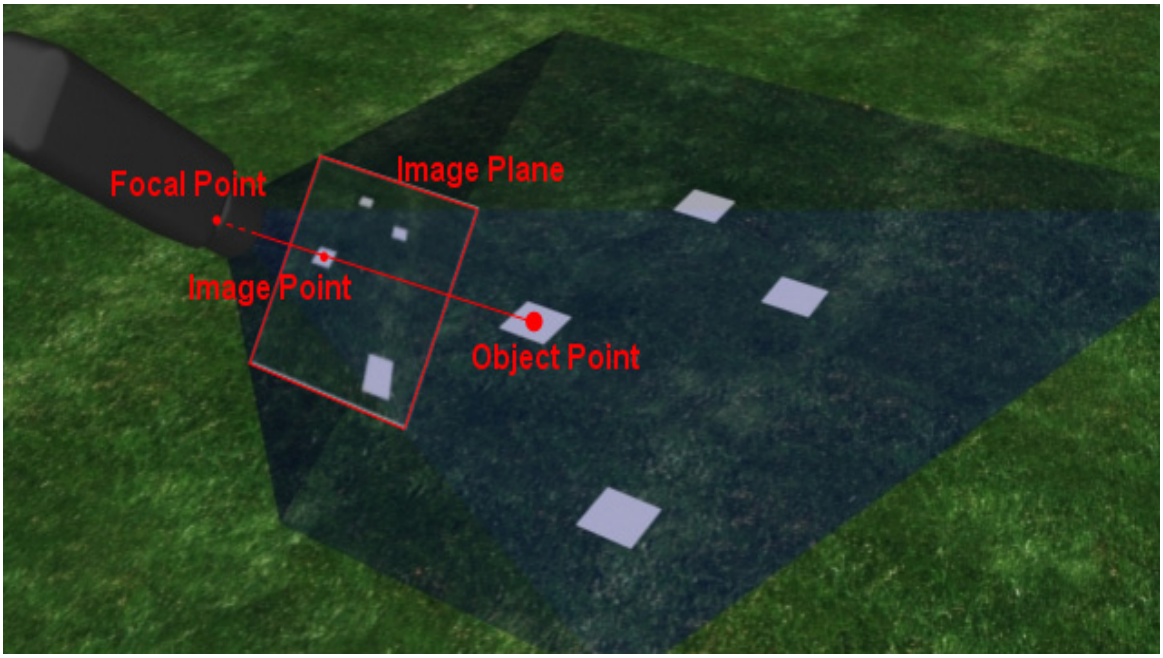

Figure 64: World-to-Image Point Transforms

Using this concept, the original location of the object can be determined from the image plane as well. Using

Figure 64 as a reference, it is obvious that using the focal point of the camera and location of the object in the image plane, a vector can be projected to show the direction of the object in space. Unfortunately, this is all that can be directly inferred from the image. The vector from the focal point to the object is known, but the exact location of the object on that vector cannot be determined from the image alone. This problem can

be solved if the vector intersects an object of known position, namely, the user defined ground plane.

Range data used in this planar positioning is nothing more than a plot of data points that reside on the pre-defined plane. These points allow an object or vehicle to be correlated to a real-world location (as defined by the WCS). To find this series of points, one of the fundamentals of projective geometry will be used; that is, the intersection of a line and a plane determine a point. The plane has already been defined, so all that is needed is a series of vectors that intersect the plane and correspond to the captured image.

Determining Pixel Vectors

A vector in space can be defined by two points. The first point is the origin of the vector; the second point determines the direction. For each vector, two points that logically relate to the camera are needed. The easiest, and most intuitive, way to accomplish this is to use the focal point of the camera and the spatial location of each pixel to create a vector. These points can be used to determine the Plücker coordinates of a line. Then using projective geometry, the intersection point of the line and the ground plane can be found.

Pixel vector points

The first point for all of the vectors will be the focal point of camera. This is a fairly intuitive selection since everything in the image converges to this single point at the time of image capture. This point is fairly easy to get and applies for all the vectors. Unfortunately, the second point of the vectors is a little more complicated to get. To aid in these calculations the camera will initially be assumed to be at the WCS origin, pointing along the positive y-axis, see Figure 65. Vector points can be repositioned to their proper place in space quite easily after they have been initially defined. These

coordinates for these vectors will be referred to as x_1, y_1, z_1 for the first point and x_2, y_2, z_2 for the second point.

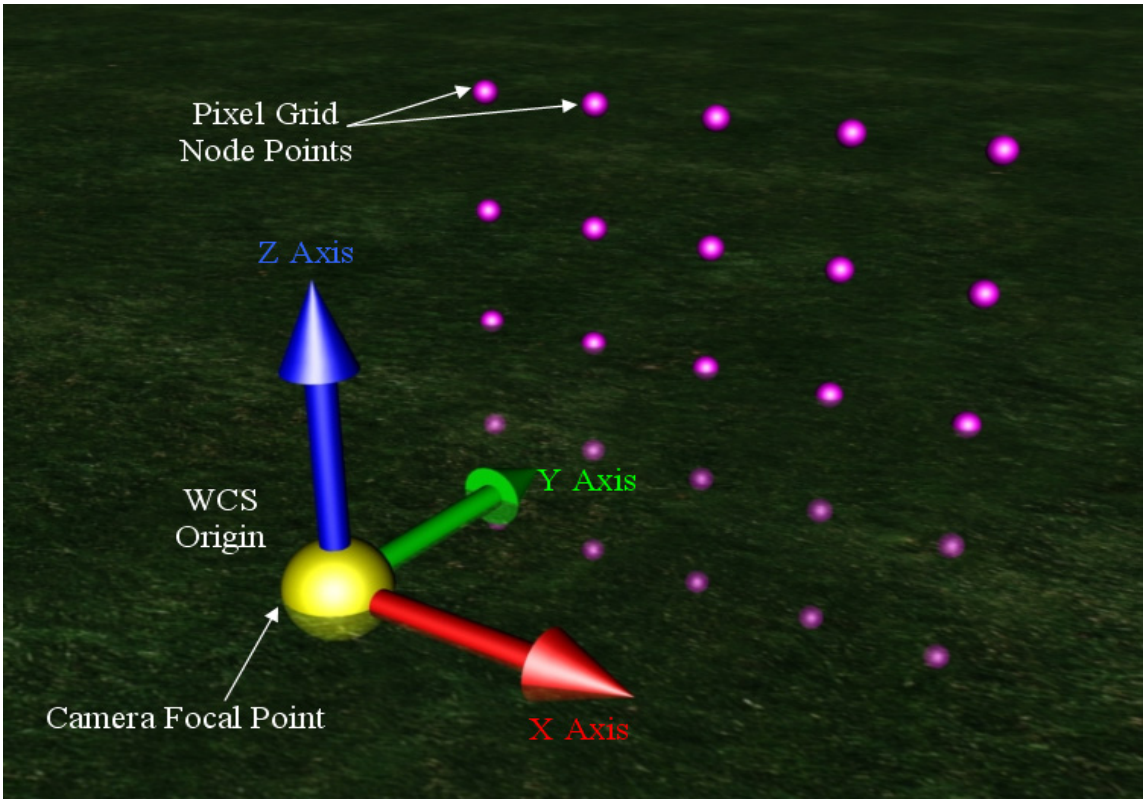

Figure 65: Initial Pixel-Grid Node Point Locations

For the first point, the values will be the same for all the vectors. The values of x_1, y_1, and z_1 will simply be the WCS origin:

$$\begin{aligned} x_1 &= 0 \\ y_1 &= 0 \\ z_1 &= 0 \end{aligned} \tag{95}$$

These variables will be the same for any camera and any capture resolution.

The second point for each pixel vector is derived from the locations of the pixels in the image plane. The image plane is a projection of scene's contents onto an arbitrary plane. This plane can be located at any location in space, but must be perpendicular to the focal axis of the camera. Placing the image plane at y = 1 will help simplify calculations.

The first order of business is to split the image plane into a pixel grid. For an image that has h x w pixels there will be (h+1) x (w+1) grid lines separating the pixels in the image. A node is defined as any location in the image plane where gridlines meet. For the same image, there will also be (h+1) x (w+1) nodes, see Figure 66. The example shown in Figure 65 and for several of the following images, depicts a sample case of the viewing volume being split into 4×4 pixels, and therefore, having 25 node points.

If the focal point of the camera is assumed to be at the origin of the WCS facing in the home direction (along the positive y-axis) and the image plane location is y = 1 then the node points can be found without using the camera's extrinsic properties. The only quantities needed are the horizontal and vertical capture resolution and the horizontal and vertical fields of view, or N_h, N_v, ψ_h, and ψ_v, respectively.

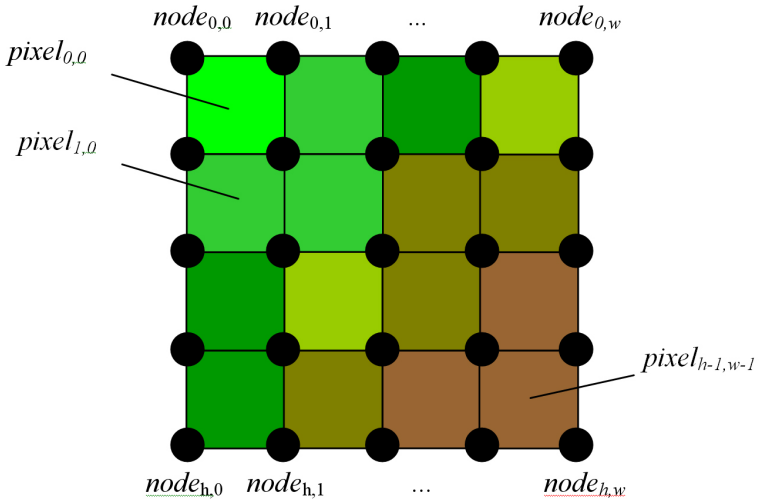

Figure 66: Sample Pixel Grid and Nodes

Every pixel takes up $1/N_h^{th}$ of the viewing volume in the horizontal and $1/N_v^{th}$ of the viewing volume in the vertical. These pixels occupy an angular fraction of the camera's FOV defined by:

$$\alpha_v = \frac{\psi_v}{N_v}$$
$$\alpha_v = \frac{\psi_v}{N_v} \tag{96}$$

Where α_h represents the horizontal angle of the image occupied by a single pixel and α_v represents the vertical. Using this quantity, the node points of the image can be found in terms of these angles. A pixel node's direction in the image from the focus is given by:

$$\alpha_v(i,j) = \frac{\psi_v}{N_v}(i) \qquad i = 0 \ldots N_v \quad j = 0 \ldots N_h$$
$$\alpha_h(i,j) = \frac{\psi_h}{N_h}(j) \qquad i = 0 \ldots N_v \quad j = 0 \ldots N_h \tag{97}$$

This establishes that $\alpha_v(i,j)$ ranges from zero to ψ_v incrementally by α_v in the vertical, and $\alpha_h(i,j)$ ranges from zero to ψ_h incrementally by α_h in the horizontal.

Since the nodes are at unit distance from the focal point, the location of the nodes in three dimensional space is found by calculating the tangent of the angle of each individual node:

$$x_2(i,j) = \frac{\tan(\psi_h)}{N_h}(j)$$
$$y_2(i,j) = 1$$
$$z_2(i,j) = \frac{\tan(\psi_v)}{N_v}(i) \tag{98}$$

Of course, it is desirable to have the image plane centered at the y-axis, so the data needs to be shifted by $-\frac{1}{2}\psi_h$ in the horizontal and $-\frac{1}{2}\psi_v$ in the vertical. Therefore the equations are modified as:

$$x_2(i,j) = \frac{2 \cdot \tan\left(\frac{1}{2}\psi_h\right)}{N_h}(j) - \tan\left(\frac{1}{2}\psi_h\right)$$
$$y_2(i,j) = 1 \tag{99}$$
$$z_2(i,j) = \frac{2 \cdot \tan\left(\frac{1}{2}\psi_v\right)}{N_v}(i) - \tan\left(\frac{1}{2}\psi_v\right)$$

All the nodes lie in a plane at $y = 1$ so y_2 will always be 1. x_2 and z_2 are dependent only on the current pixel index, resolution of the image, and the FOV of the camera. Therefore, all of the preceding equations are independent of camera location. By calculating this information first, it will not have to be recalculated when the camera is moved or rotated. The methodology that is used to create these vectors may seem a bit awkward, but has proven to be relatively quick and very reliable for finding appropriate range data points.

Reorienting pixel vectors

The relationship between the pixel grid and focal point do not change for a given camera and capture resolution, regardless of what the camera is doing. This feature allows these original correlations to remain unchanged for any camera position, see Figure 67. The objective at this point is to get the focal point and pixel grid into the proper location as determined by the location of the camera. Instead of calculating a new image plane and pixel grid, the coordinate system can be moved instead. Since all the

vectors originate from a single point, this point can simply be translated to its proper location in space. So x_1, y_1, z_1 are translated to the location of the focal point of the camera in space by dx, dy, dz:

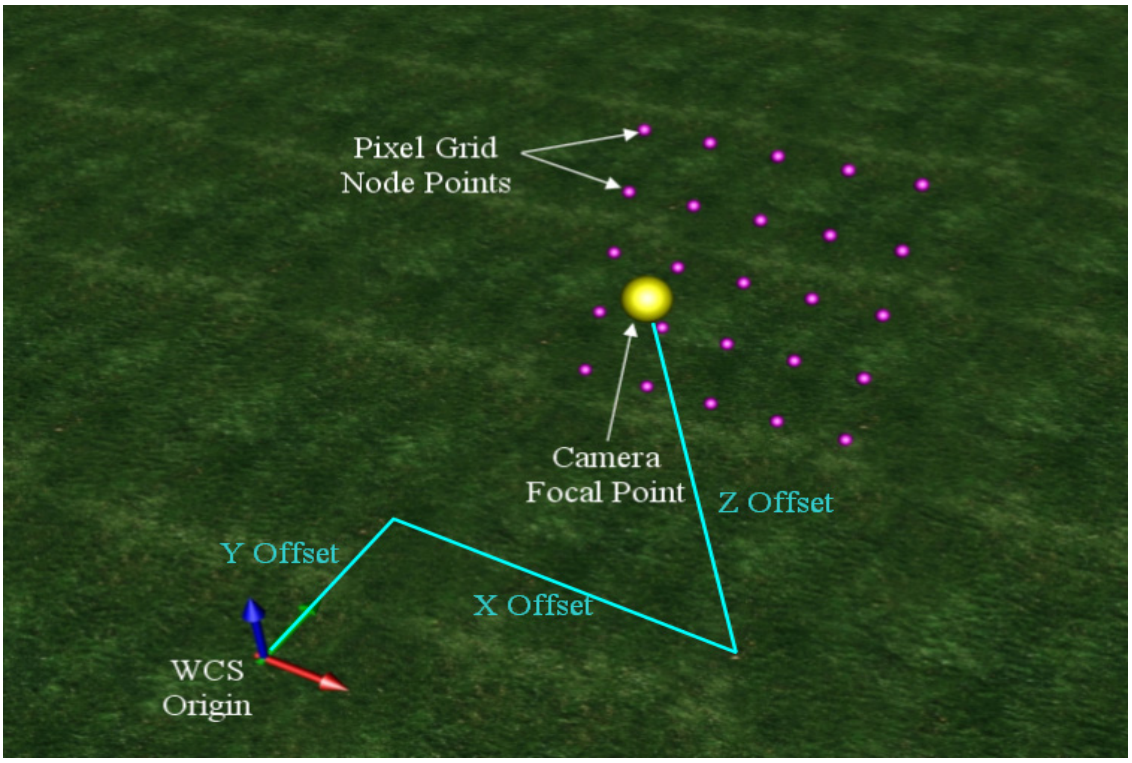

Figure 67: Pixel-Grid Node Point Locations after Translation and Rotation

$$\begin{aligned} x_1 &\leftarrow x_1 + dx \\ y_1 &\leftarrow y_1 + dy \\ z_1 &\leftarrow z_1 + dz \end{aligned} \tag{100}$$

The gridline nodes must be translated and rotated about the focal point depending on the position and orientation of the camera, so all of the remaining points must be fully converted. This can be done using the combination of translation and rotation matrices described in Equation (94). Each point is multiplied by the matrix $\overline{\overline{R}}$:

$$\begin{bmatrix} x_2(i,j) \\ y_2(i,j) \\ z_2(i,j) \\ 1 \end{bmatrix} \leftarrow \overline{\overline{R}} \cdot \begin{bmatrix} x_2(i,j) \\ y_2(i,j) \\ z_2(i,j) \\ 1 \end{bmatrix} \tag{101}$$

This step requires that $\overline{\overline{R}}$ be calculated once and then applied to each pixel node through matrix multiplication. For a standard size image (640x480) there will be 641x481 nodes, or 308,321 total nodes. It's apparent that this step will take a significant amount of time.

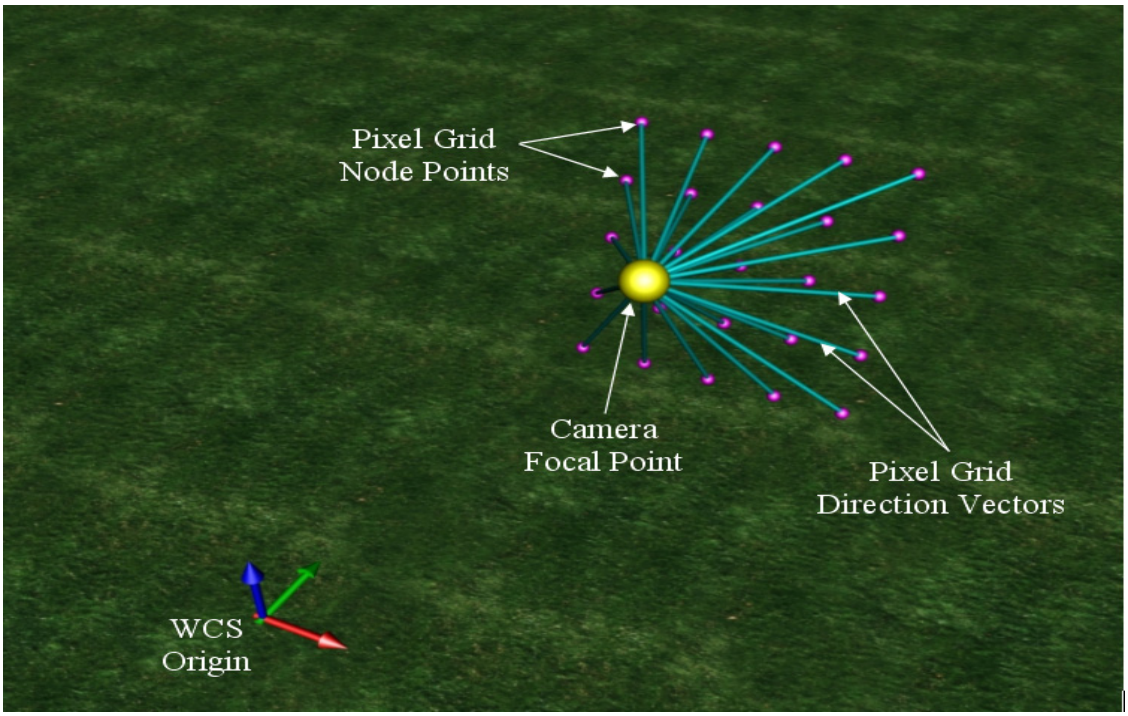

Figure 68: Pixel Node Vectors

Creating pixel node vectors

Now that the focal point and pixel grid is in the proper location, the vectors must be created. The vectors must be defined in terms of Plücker coordinates. The calculations to do this are fairly simple, but again, must be done for hundreds of thousands of nodes. Figure 68 shows an example of the pixel node vectors.

As defined in earlier sections of this chapter, Plücker coordinates for a line are displayed as {L,M,N;P,Q,R}. The definitions for L, M, and N can be found in Equation (71) and the definitions for P, Q, and R can be found in Equation (79). Redefining the

line equation in terms of the focal point and pixel grid coordinates and grouping them in terms of $\{\overline{S_1}, \overline{S_{01}}\}$ yields:

$$\left\{ \begin{bmatrix} \dfrac{x_2(i,j)-x_1}{|S(i,j)|} \\ \dfrac{y_2(i,j)-y_1}{|S(i,j)|} \\ \dfrac{z_2(i,j)-z_1}{|S(i,j)|} \end{bmatrix} ; \begin{bmatrix} \dfrac{y_1(z_2(i,j)-z_1)-z_1(y_2(i,j)-y_1)}{|S(i,j)|} \\ \dfrac{z_1(x_2(i,j)-x_1)-x_1(z_2(i,j)-z_1)}{|S(i,j)|} \\ \dfrac{x_1(y_2(i,j)-y_1)-y_1(x_2(i,j)-x_1)}{|S(i,j)|} \end{bmatrix} \right\}_{i,j} \tag{102}$$

Where $|S(i,j)| = \sqrt{(x_2(i,j)-x_1)^2 + (y_2(i,j)-y_1)^2 + (z_2(i,j)-z_1)^2}$.

Defining the ground plane

The ground plane will be defined in the homogeneous format that was described as [D;A,B,C] by Equation (67), earlier in this chapter. This can be rewritten in the form:

$$[D_2; \overline{S_2}] \tag{103}$$

where $\overline{S_2}$ is equivalent to the normal vector *Ax+By+Cz* and D_2 is the normal distance to the plane from the origin. The most common plane that will be used is the level ground plane which has a homogeneous planar equation of [0;0,0,1] indicating that the plane is perpendicular to the z-axis and coincident with the origin.

Creating a range point

The coordinates of a point can be shown homogeneously, as shown in Equation (59) on page 113. This format can be rewritten as:

$$(w_3; \overline{S_3}) \tag{104}$$

Where $\overline{S_3}$ is the vector notation of the x, y, and z coordinates of the point. This new notation allows for a simple equality to be set with Equation (88) on page 118:

$$\left[\overline{S_2}\cdot\overline{S_1};\overline{S_2}\times\overline{S_{01}}-D_2\overline{S_1}\right]=\left(w_3;\overline{S_3}\right) \tag{105}$$

This can be expressed by the following two equations:

$$\begin{aligned}\overline{S_2}\cdot\overline{S_1}&=w_3\\ \overline{S_2}\times\overline{S_{01}}-D_2\overline{S_1}&=\overline{S_3}\end{aligned} \tag{106}$$

In normal space w_3 should be positive and non-zero. If $w_3\leq 0$, the range value will be incorrect. There are two special cases that must be checked for at runtime to confirm that the range value obtained is viable. These will be discussed shortly. For now w_3 is assumed to be greater than zero.

A convenient feature of projective geometry is the coordinates of an object are homogeneous. This means that a point with coordinates of $(w;x,y,z)$ is equivalent to $(\lambda w;\ \lambda x,\ \lambda y,\ \lambda z)$. Using this property, the w can be divided out of the coordinates:

$$\left(1;\frac{x}{w},\frac{y}{w},\frac{z}{w}\right) \tag{107}$$

Combining Equations (105) and (107) and expanding yields:

$$\overline{N}=\left(\frac{\overline{S_3}}{w_3}\right)=\frac{\left[\overline{S_2}\times\overline{S_{01}}-D_2\overline{S_1}\right]}{\overline{S_2}\cdot\overline{S_1}} \tag{108}$$

The new variable, $\overline{N}$, represents the coordinates of the new point. This equation can be used to project the array of pixel grid nodes onto the plane. Updating the equation to account for the array of pixel grid nodes yields:

$$\overline{N(i,j)} = \frac{\left[\overline{S_2} \times \overline{S_{01}(i,j)} - D_2 \overline{S_1(i,j)}\right]}{\overline{S_2} \cdot \overline{S_1(i,j)}} \tag{109}$$

where the new quantity $\overline{N(i,j)}$ represents every node vector's intersection with the plane. Figure 69 displays the creation of range data points.

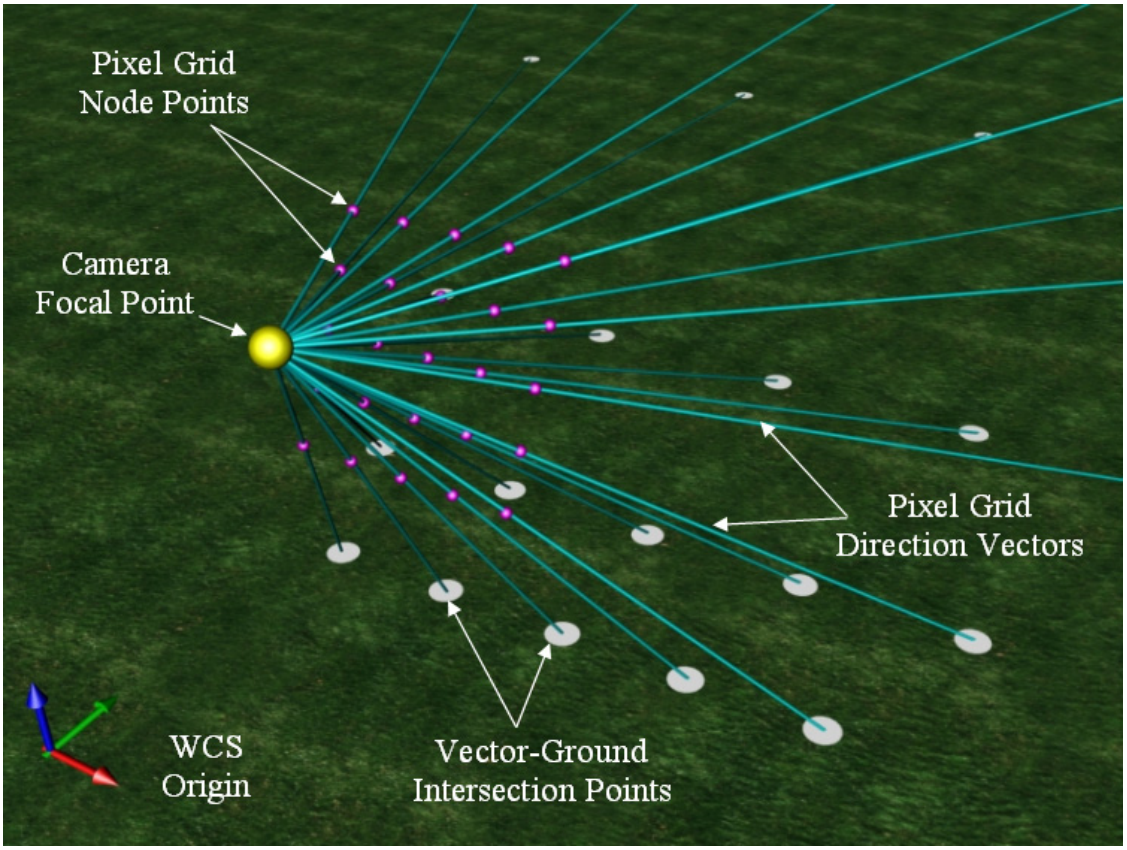

Figure 69: Node Vector Intersection Points

Creating pixel areas

Each of the projected node points, $\overline{N(i,j)}$, corresponds to a corner of what will referred to as a pixel area, from here on. This pixel area is the projection of the pixel onto the ground plane. All of the color in this area is integrated to create the resulting pixel color for that particular pixel. The shape and size of these areas vary depending on the type of lens, distance from the camera, orientation of the camera, etc. The pixel area, while dynamic in nature, will always be a quadrilateral.

Even though a pixel represents a measurable area in the scene, it is more convenient to imagine the pixel as representing a single point. Any object that is captured in a pixel must be assumed to be in a fixed location with a probability of error. The centroid is the intuitive point to assume the location of the pixel area. Since the area is recognized to be consistent, the centroid can be found by simply averaging the corner points for each pixel area, see Figure 70.

$$P(i,j) = \frac{N(i,j) + N(i+1,j) + N(i+1,j+1) + N(i,j+1)}{4} \tag{110}$$

The maximum error associated with each pixel can be found by determining the maximum distance between the centroid of the pixel and each of the corners.

$$E(i,j) = \max\begin{cases} abs(P(i,j) - N(i,j)) \\ abs(P(i,j) - N(i+1,j)) \\ abs(P(i,j) - N(i,j+1)) \\ abs(P(i,j) - N(i+1,j+1)) \end{cases} \tag{111}$$

The error, *E(i,j)*, represents the radius of a sphere of error. Defining the error in this way will typically make error values much higher than they actually are, but this method is still preferable to defining error for all axes independently. *E(i,j)* symbolizes a worst case error estimation and is the maximum error that can be accrued from condensing the pixel area to a single point. The error incurred from poorly defined camera values is not included in this equation. Measurement errors must be found empirically, not quantified through an equation.

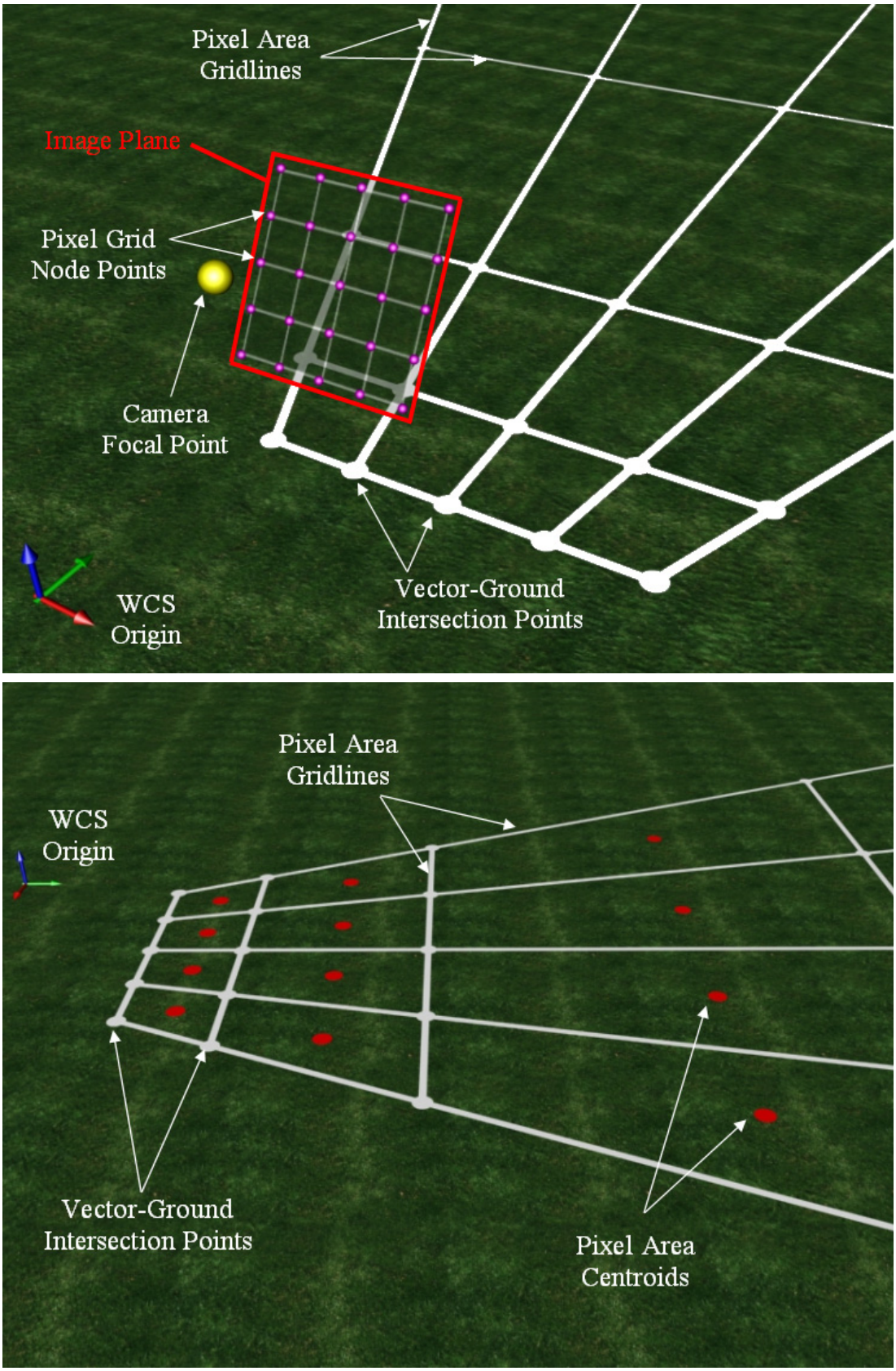

Figure 70: Pixel Area Gridlines (Top). Pixel Centroid Locations (Bottom)

Creating look-up table

Using Equations (110) and (111) the location and error can be found for every pixel in the image. These locations are calculated once for a camera at a fixed location. The data only changes if the camera is moved, so with static cameras, the range data can be calculated once and put into a look-up table (LUT). Once the data is saved to the LUT, the image can be scanned by the processing algorithms. If a feature is found to reside in a certain pixel, the location of this pixel can be used to query the LUT and find the pre-calculated range information for this pixel.

Potential Problems

The pixel point range data calculations work well as long as the viewing volume extends completely into the ground plane. This ideal situation is not always present though. If the camera is facing a direction where the horizon can be seen or the ground plane cannot be seen, there are a few inherent problems that can occur. All the potential problems can be detected by examining the w_3 quantity.

Points at infinity

If w_3=0, then the line's directional component $\overline{S_1}$ and the normal vector for the plane $\overline{S_2}$ are perpendicular. This indicates that the line and the plane are parallel. In this instance the plane and line will intersect at a point at infinity. Any node with a w_3 value of zero will indicate that the particular node is located infinitely far away. These nodes must be discounted along with the pixels areas that the node is attached to. The occurrence of infinite values is rare because of the precision used in calculating these variables. Typically, w_3 will be very close to zero and therefore create a range point that

can be millions of units (inches, feet, meters, etc.) away. When this happens, the point is considered distant. This will be covered in a later section.

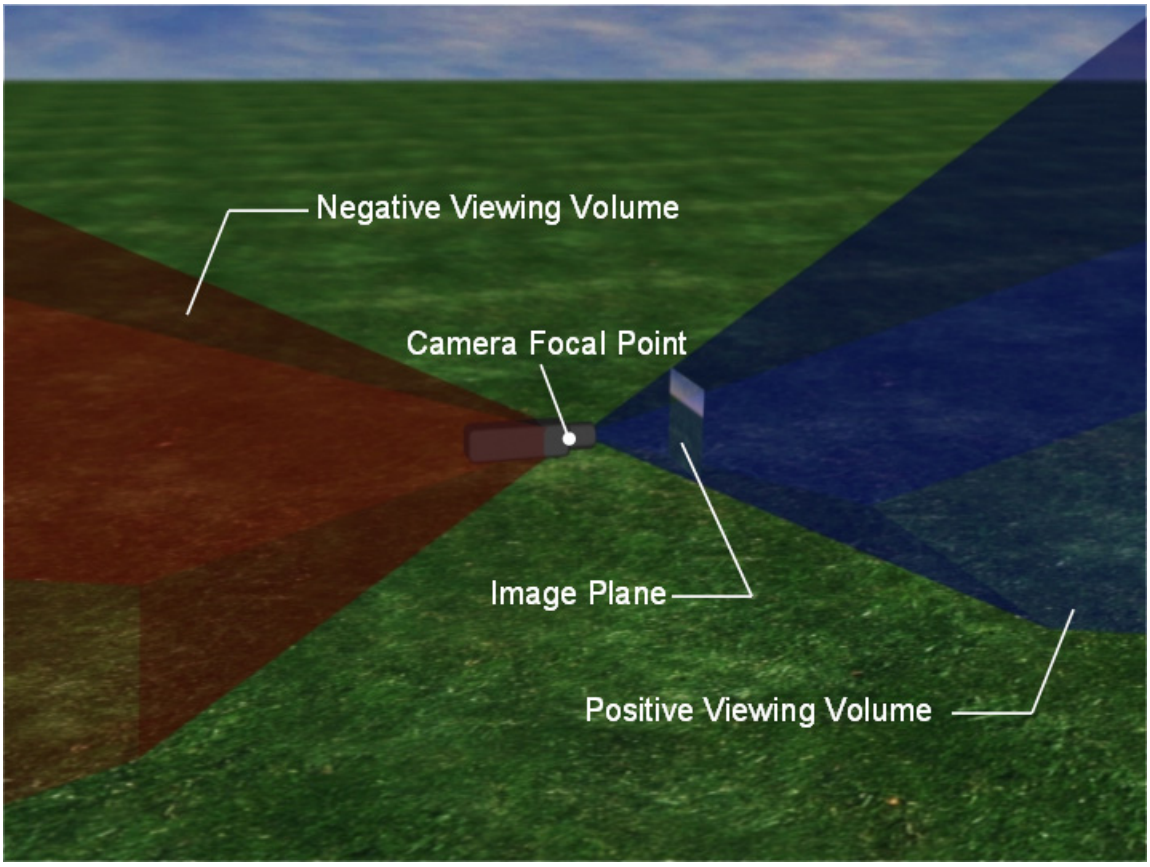

Figure 71: Positive and Negative Viewing Volumes

Negative points

The other case that will cause an error is much more common. A line extends infinitely in the positive and negative directions of its vector. Because of this the viewing volume has what can be referred to as a *negative viewing volume*, see Figure 71. Any node vectors that do not intersect the plane inside the viewing volume and do not run parallel to the plane, will intersect inside the negative viewing volume. This occurs when the camera is pointed toward or above the horizon, making $w_3<0$ for many of the vectors. The line and plane intersection equation finds an intersection regardless of where it occurs, even if the intersection is in the negative direction of the vector. As with the

infinite node points, pixel areas with negative intersection nodes must be removed. Otherwise, the algorithm would think that the area behind the camera is inside the viewing volume.

Distant points

The final scenario is when pixel nodes are calculated to be at distances that are very far from the camera. The range resolution decreases exponentially as the distance increases. Therefore, it may be useful to clip these areas from the LUT or, preferably, reorient the camera so that it points more directly at the plane. The selection of the maximum range is somewhat subjective.

The actual range that is considered distant will change from one application to the next. For example, if a toy car is being tracked, the maximum range distance may only be a few feet before the vehicle is far enough away to make visual tracking inadvisable. On the other hand, using aerial reconnaissance to track the movements of a tank could have a maximum range distance of several thousand feet.

For any application, the required accuracy should be checked against the maximum error for each pixel. If the max error exceeds the required accuracy, the data from the visual positioning system may not be useable. For most cases, swapping cameras and/or video capture devices for better, higher resolution counterparts will solve any accuracy problems that arise.

CHAPTER 5
PLANAR VISUAL POSITIONING APPLICATION

Now that a method for creating a pixel data lookup table has been defined, the concept needs to be expanded to function as a positioning system. A few key elements must be present to effectively use this information to track a vehicle. First, a world coordinate system must be defined. Second, one or more cameras must be set up to view the environment that the vehicle is to be in. A vehicle coordinate system must be defined, and tracking features must be placed on the vehicle so that the software can distinguish the vehicle from its surroundings. Finally, precise measurements of all the preceding items must be made.

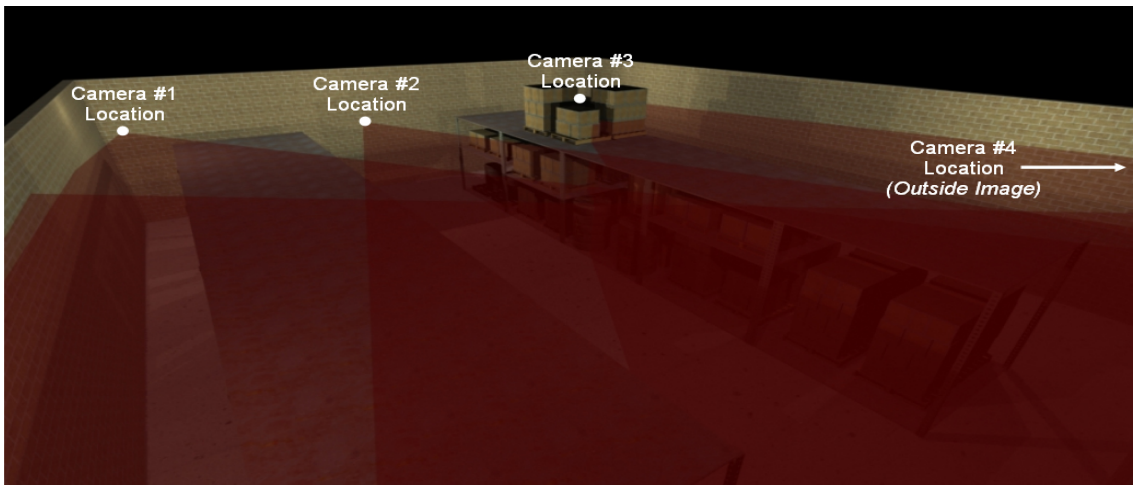

Figure 72: Example Camera Placement

When all of these things are defined and measured, a useable positioning system can be created. Caution should be used in these initial stages since it will make subsequent work much less complicated. Measurements of the environment, camera

placement, vehicle features, etc. should be carefully made because the accuracy of the positioning system is directly proportional to the accuracy of these measurements.

Building the Environment

This type of position system can be used for many types of environments, but it lends itself particularly well to large indoor applications like warehouses. The only real requirement is that the surface that vehicle is driving on is fairly flat. The surface does not need to be horizontal; it can be a planar region at any incline, as long as it's consistent.

The first order of business is to declare a world coordinate system, or WCS. The WCS can be at any position or orientation, but it would definitely suit the user to orient it with the z-axis perpendicular to the ground plane. This way, the vehicle coordinates will be in terms of x and y with a constant z component. It is much easier to visualize vehicle position when only two terms are involved, but the system will still work if a more complicated setup is desired.

Camera Placement

Cameras should be placed around the environment in a fashion that one or more cameras can always clearly view the vehicle's tracking features, see Figure 72. It is obvious that if a camera cannot clearly see the vehicle, there will be no location information present about the vehicle at that time. It is always better to have too many cameras than to not have enough.

Each camera's position in the WCS must be known with reasonable accuracy. The camera's xyz placement should be measured from the focal point of the camera. The focal point is the area directly behind the lens where the light focuses on the capture element (typically a CCD array). Although this array has a finite area, it can be assumed

that the camera focuses to a point. The selection of this point will be approximate, but minor deviations will make little difference.

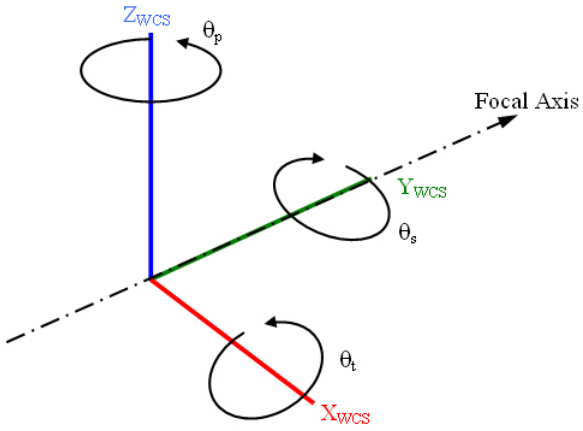

Figure 73: Initial Camera Alignment with the WCS

The angular extrinsic values of the camera will be defined relative to the selection of the coordinate system. In a typical system, where z is positive upward, the camera pan angle will be positive counter-clockwise. The tilt angle will be positive upward, and slant angle will be positive clockwise about the focal axis, see Figure 73.

The intrinsic values of the camera will vary from camera to camera. If the camera has zoom capabilities, it is recommended that camera be zoomed completely in or out. It is difficult to tell the exact magnification of the zoom when it is not at its maximum or minimum. Since the field of view properties are directly related to the zoom factor, not knowing the exact zoom can significantly compromise the rest of the position system. Typically the focal length of a given camera can be found in the owner's manual or on the internet. If the fields of view cannot be found for the camera, Appendix B has more information on how to get this information.

The capture resolution in the horizontal and vertical (N_h and N_v, respectively), are set by the video capture device. Higher capture resolutions give more accurate data about

a scene, but take longer to process on a frame-to-frame basis. Lower resolutions are less accurate, but faster to process, so the desired capture resolution depends on the situation.

Vehicle Coordinate System

Every vehicle is different and even the features that could be used on the same type of vehicle can differ from one situation to the next. For this reason, it is preferable to relate the position of vehicle features to a vehicle coordinate system, or VCS. By allowing the vehicle to have its own coordinate system, discrepancies from one vehicle to the next can be handled relative to the VCS. This way the effects of changing vehicles or tracking features are transparent to the tracking system. Position and orientation of the vehicle is determined by the location and direction of the VCS, not by the features themselves.

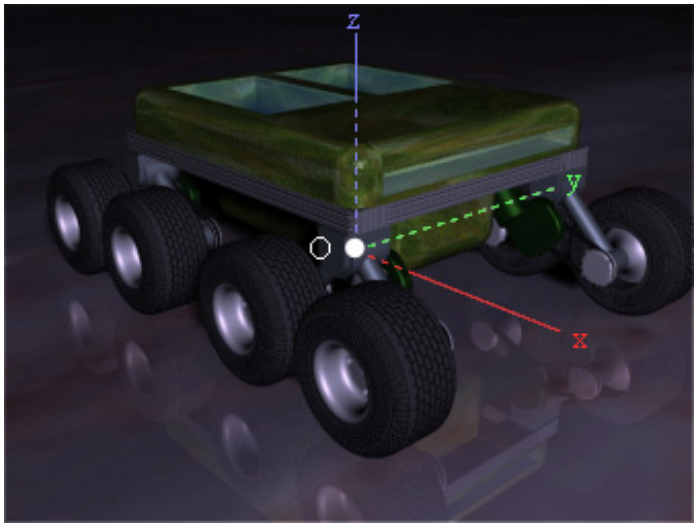

Figure 74: Sample Vehicle Coordinate System

The VCS should be a point somewhere on or around the vehicle. A good place to define the VCS is at the ground level in the center of the vehicle, see Figure 74. The position of the VCS is arbitrary, but the orientation must be set so that the x-axis is pointing to the forward direction of the vehicle. When the vehicle is moving in a purely forward direction it will be considered to be moving along the positive direction of the VCS x-axis. With this setup, the vehicle's orientation can be found by determining the direction of the VCS's x-axis with respect to the WCS.

The location of the vehicle in the WCS will be given as a single point; this point will be the location of the origin of the VCS in the WCS space. Therefore, it is important to choose a position for the VCS on the vehicle that will be the most beneficial for the required application. For instance, let's say the vehicle is being tracked while traveling along a typical road. In this situation it is imperative that the vehicle not venture into oncoming traffic, so it may be more beneficial to place the VCS on the left side of the vehicle, so precise measurements of the vehicle's driver's side can be closely tracked. If the vehicle is to be traveling indoors with obstacles randomly placed, it will be best to have the VCS at the vehicle center.

Vehicle Tracking Features

The features on the vehicle that will be used for tracking must fit a few requirements. First, they must reside in a plane parallel to the defined ground plane. For most vehicles, this will entail putting the features on the vehicle's roof, see Figure 75. The features used in this example are red and blue lights placed on top of the vehicle. The tracking plane is indicated red translucent polygon.

The second requirement for the vehicle's tracking features is for them to be distinguishable from the rest of the image. The red and blue lights used as features in Figure 75 were utilized because of their relative dissimilarity with

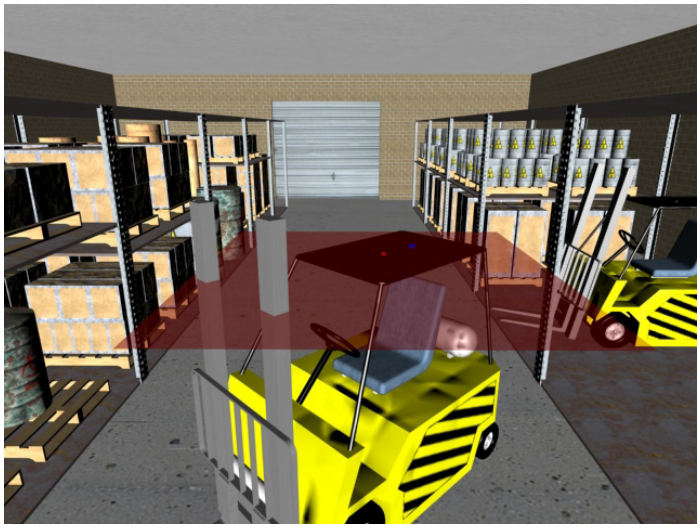

Figure 75: Planar Vehicle Tracking Features

common colors in the image. While there is a little bit of red present in the background, it is rare and significantly darker than the red of the tracking feature. The blue feature is also a much different shade than any other blue in the image. This makes the features discernible from the rest of the environment.

There is not always a color that can be selected that will completely isolate the feature colors from data in the rest of the image. To compensate for this, newer tracking features include a foreground and background color. The foreground color represents the actual color of the feature. The background color represents the color of the areas directly surrounding the features. To increase accuracy, the foreground color for each feature is searched for; then the edges of the feature are searched to find if any of the background color is present. A feature foreground that is bordered by many background pixels is much more likely to be the actual tracking feature.

When using fairly unique foreground and background colors for the features, the chances of a false hit drop dramatically. Figure 76 shows the new concept of vehicle roof features. Feature 1 is a bright shade of green and feature 2 is a bright shade of cyan. The background color used for both features is a navy blue. It's clear from Figure 76 that there are other artifacts in the image that share a common color palette with feature 1. In processing this image, the use of background colors will be essential to being

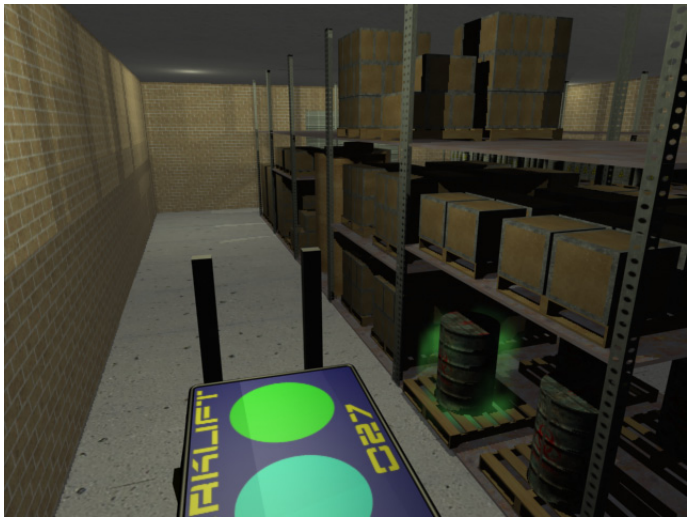

Figure 76: Updated Vehicle Tracking Features

able to distinguish between this feature and ambient green in the image.

The third requirement is that the features must be large enough to clearly see from the camera. This new requirement was added along with the inclusion of background colors and can be seen in Figure 76, as well. In these more recent simulations the forklift's roof has been painted with large radius circles so that the features are clear even at a distance.

The fourth requirement is that the features must be visible in any type of lighting condition that could be present in the given environmental conditions. Figure 77 shows the results of poor feature selection in non-optimal lighting conditions. This image was taken in a trial run in a local area warehouse. The features atop the vehicle are fluorescent yellow and pink on top of a fluorescent green background. These colors were chosen to ensure that they stood out from the colors present in the warehouse. Unfortunately, the combination of an inexpensive video camera and poor lighting conditions made these features wash out and appear completely white. Conditions like this require that extra thought be put into feature selection.

Figure 77: Poor Feature Visibility due to Bad Lighting Conditions

The final requirement of the tracking features is to know their positional relationship to the vehicle and to each other. The centroids of the features must be

measured in xyz coordinates from the VCS origin. The centroids of the features are defined as points $\overline{F_1}$ and $\overline{F_2}$ for feature 1 and feature 2, respectively. Ideally, feature 1 would be towards the front of the vehicle in the middle and feature 2 would be directly behind it. This way the vector created by the difference of these two points would be pointing along the VCS x-axis, facilitating the location of the vehicle orientation. This may not always be the case so first a feature vector must be calculated:

$$\overline{FV} = \overline{F_1} - \overline{F_2} \tag{112}$$

If the features are set in the ideal configuration, $\overline{FV}$ will equal [n,0,0]. Where n is the distance between feature 1 and feature2. Under the non-ideal configuration, $\overline{FV}$ will be an arbitrary vector representing the orientation of the features with respect to the VCS.

The tracking features of the vehicle must be coplanar and parallel to the surface that the vehicle is driving on, therefore, it can be seen that the features must lie in the x-y plane in the VCS. For the sake of determining the vehicle's orientation the z-component of the features can then be dropped since they must be equal for both features. Then, by using the definition of a dot product:

$$\cos(\theta) = \frac{V \cdot W}{|V| \cdot |W|} \tag{113}$$

the angle between the $\overline{FV}$ and the VCS x-axis can be found:

$$OCA = \cos^{-1}\left(\frac{\overline{FV} \cdot \hat{x}}{\left|\overline{FV}\right|}\right) \tag{114}$$

The variable $\hat{x}$ represents a unit vector in the direction of the x-axis, and *OCA* represents the 'Orientation Correction Angle'. When the vehicle orientation is found in the image, the *OCA* must be subtracted from the rotational angle about the z-axis to determine the correct orientation. The OCA can be positive or negative depending on the direction the front feature is offset from the back. If the front feature is rotated clockwise from the rear feature, the OCA should be positive. If the front feature is offset counterclockwise from rear feature, the OCA should be negative.

Measuring the Environment

As stated before, accurately measuring the environment is essential to achieving good tracking results. All possible areas that the vehicle can travel in should be measured with respect to the WCS. In addition to the floor plan of the environment, any obstacles can be mapped as well. This information can be used to clip areas from the map that the vehicle cannot travel in.

Tracking a Vehicle with Visual Position System

Once the environment has been measured and the vehicle has been prepped, the difficult part is over. The positioning system can now use this information to visually track a vehicle. The software runs in stages so that functions can be grouped to calculate more efficiently.

Program Initialization

The first stage of computation is the program initialization stage. This part of the program is fairly lengthy, but only needs to be done once. In this phase, the input and output files are opened, the lookup table (LUT) is generated, problematic and user-defined areas are clipped, the overlays are built, and feature tracking information is defined.

Open I/O files

The first process is to open the input and output files. This is done at the beginning of the program initialization for two reasons. First, it is inefficient to open the files, read/write data, and close the file every time data needs to be processed. The files are opened in initialization and left open until the program is de-initialized. Second, vital information about the video stream is read from the headers of any input files. This data is used to determine the capture resolution of the image, length of the stream, color depth, etc.

Generate lookup table

Once the inputs have been initialized, the LUT can be generated. This is perhaps the longest process of the program because of the vast number of complex calculations that must be made. For every node in the image, the original node location must be calculated, the node must be rotated and translated to the proper location in space, a line must be generated between the focal point and the node, the intersection of the line with the ground plane must be found, the nodes must be averaged to find the center of each pixel, and the maximum error between the average point and each node must be found. Even a small video stream with a resolution of (320 x 240) contains 76,800 pixels and therefore has 77,361 nodes. All of these things must be done for each node. It's easy to see that with a typical stream that is (640 x 480) there is a total of 308,321 nodes and quite a few more calculations to be done. It is, therefore, imperative that this calculation only be done once.

Remove erroneous pixels

After the LUT is generated, erroneous nodes and pixels must be stricken from the table. Erroneous nodes are those which lie at infinity or in the negative viewing volume.

This can be checked for by ensuring that the value of w_3 is greater than zero. If a pixel is found that has a negative or null w_3 value, that node is then set to be a null pixel and is discarded from the LUT.

Remove clipped areas

The next order of business is to remove areas that defined by the user to be out-of-bounds. Clipped areas are handled in the same manner as erroneous areas, so no tracking information is present for a pixel when it is clipped. If no clipping areas are defined then all viable locations will be kept. The user has the option to clip areas by indicating a series of include and exclude polygons. The data is sent to the program as a string; the control characters are listed with their functions in Table 6.

Table 6: Clipping Options

Character String	Description
a	Autoclip area *(Have the program decide a clip region on its own)*
i	Include polygon
e	Exclude polygon
c	Clip all pixels
u	Unclip all pixels

Include and exclude operations are performed in the sequence that they are present in the string. Therefore, partial or total areas added or subtracted by a polygon can be reversed by a subsequent polygon. This allows for almost any configuration to be properly set for tracking. An example of a clipping string would be:

c i (0,0 ; 0,10 ; 10,10 ; 10,0) e (1,1 ; 1,2 ; 2,2 ; 2,1)

This string indicates to initially set all the pixels as out-of-bounds. Next, it includes a 10×10 square to the available area (originating at the origin of the WCS). Finally, it excludes a 1×1 square inside this area as out-of-bounds.

Create overlays

At this point the LUT is finalized. All pixels that need to be clipped have been removed from the table. With this new array of values the overlays can be set. There are a few different overlays that can be added by the user. The list of possible overlays is shown in Table 7, with the control character listed on the left and the type of overlay listed on the right.

Table 7: Overlay Options

Character String	Description
x	Display X – Gridlines
y	Display Y – Gridlines
z	Display Z – Gridlines
r	Display Range Lines *(Normal distance from the camera to the point in space)*
b	Remove boundary areas.

In the case of the gridlines and range lines, the user specifies how many lines to have of each type. The program then searches for the maximum and minimum x, y, and z values in the LUT (excluding those which were already clipped) and uses this information to place equidistant gridlines in the image. Then the LUT must be scanned to determine the closest possible range matches for each line. That pixel is then tagged to show the appropriate grid information. Also, the range value is displayed in the image at a point somewhere on each gridline. The final overlay, remove boundary, searches the

LUT for pixels that have been marked as null. The pixel is then displayed as a user-defined color when shown on the screen. The net effect of this is that only the areas that the vehicle can travel in will be shown in the display windows.

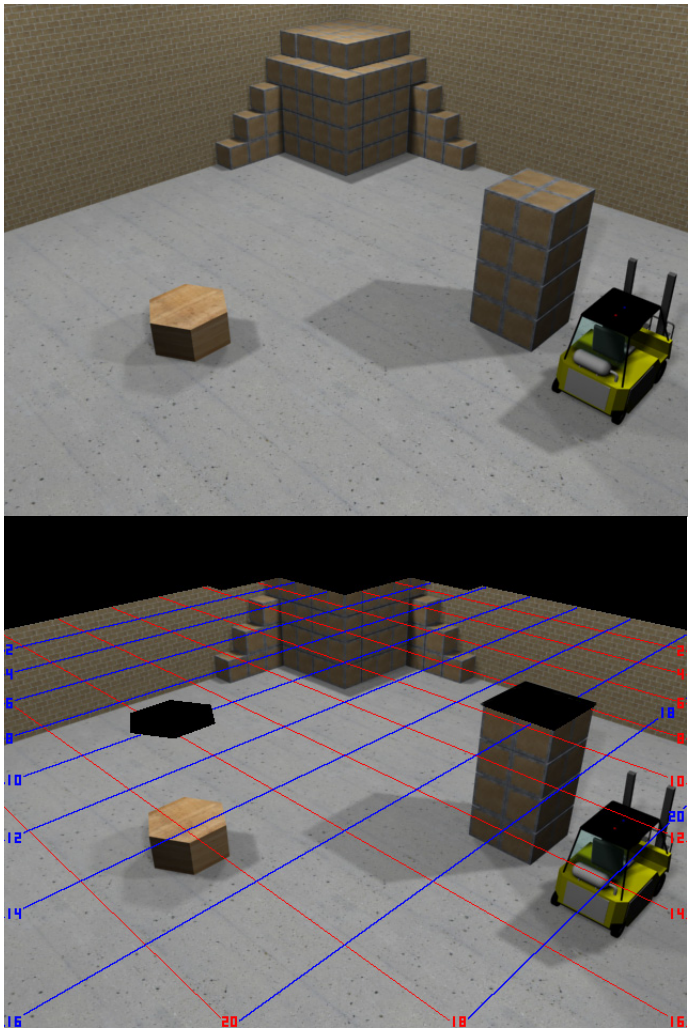

Figure 78: Original Image (Top). Clipped Image with Grid Overlay (Bottom).

Figure 78 shows an example warehouse simulation image. The top picture is a still image from a camera in one of the corners of the warehouse. The walls, along with the boxes in the image, have been mapped into the clipping parameters in world coordinates and input to the software. The bottom image shows the visual output from the position system. X coordinate grid lines are shown in red, y grid lines are shown in blue, and areas occupied by obstacles are shown in black. The position grid is shown at a height of several feet off of the ground to avoid covering the obstacles and walls with the black overlays.

Define feature tracking information

The final phase of program initialization is the determination of feature tracking data. This information must be defined from multiple sets of training data and placed

into a file. This file will contain the relevant tracking information about a set of features. Retrieving this information from a file instead of user input is convenient when dealing with several different sets of features. Instead of entering numerous statistical values into the program this information can be read directly from the pre-defined tracking data files.

These tracking data files can be manually created by the user or automatically generated with a separate program that determines statistical data from a set of training images. This program is discussed further in Appendix C. The tracking data is in the American Standard Code for Information Interchange, or ASCII, format. Using the ASCII format instead of binary, the user is able to inspect and tweak tracking data values by simply modifying text values.

Every tracking data string starts with a five character information header. The first character indicates the feature number. This must be a '1' or a '2'. The second character indicates if the value corresponds to the feature's foreground or background. This character must be a 'f' or a 'b'. The next three characters are op-codes for different variables present in the foreground and background of each feature. A list of the codes is shown for each type of distribution used in this program (3d Gaussian, 2d normalized Gaussian, color direction, and color range) separately the following tables.

The Mahalanobis Distance must be calculated to determine if a pixel is inside the ellipsoid of a three-dimensional Gaussian distribution. To calculate the Mahalanobis Distance, the ellipsoid's mean vector and inverted covariance matrix must be known:

Table 8: Tracking Data Op-Codes for Gaussian Distribution

Character String	Description
rme	Feature's Red Color Component Mean
gme	Feature's Green Color Component Mean
bme	Feature's Blue Color Component Mean
irr	Red – Red Element of the Inverted Covariance Matrix
igg	Green – Green Element of the Inverted Covariance Matrix
ibb	Blue – Blue Element of the Inverted Covariance Matrix
irg	Red – Green Element of the Inverted Covariance Matrix
irb	Red – Blue Element of the Inverted Covariance Matrix
igb	Green – Blue Element of the Inverted Covariance Matrix
3dd	Maximum Mahalanobis Distance to Include in Distribution

For the normalized Gaussian distribution, the colors are represented by two-dimensional coordinates. Like before, the Mahalanobis Distance must be calculated. Again, the mean and inverted covariance matrix must be known:

Table 9: Tracking Data Op-Codes for Normalized Gaussian Distribution

Character String	Description
	Feature's Planar X-Component Mean
yme	Feature's Planar Y-Component Mean
ixx	X – X Element of the Inverted Covariance Matrix (for Normalized Gaussian)

Table 9. Continued

Character String	Description
ixy	X – Y Element of the Inverted Covariance Matrix (for Normalized Gaussian)
iyy	Y – Y Element of the Inverted Covariance Matrix (for Normalized Gaussian)
2dd	Maximum Mahalanobis Distance to Include in Distribution

For color direction, the mean color vector of the image must be known along with the amount of variance allowed in each color direction:

Table 10: Tracking Data Op-Codes for Normalized Color Direction

Character String	Description
rdi	Normalized Red Directional Component of Color Vector in RGB Space
gdi	Normalized Green Directional Component of Color Vector in RGB Space
bdi	Normalized Blue Directional Component of Color Vector in RGB Space
rva	Variance of the Color Vector in the Red Direction in RGB Space
gva	Variance of the Color Vector in the Green Direction in RGB Space
bva	Variance of the Color Vector in the Blue Direction in RGB Space

The lowest level search, simple color range, only the red, green, and blue's high and low values are needed:

Table 11: Tracking Data Op-Codes for Simple Color Range

Character String	Description
rhi	Maximum Red Pixel Value (Integer)
ghi	Maximum Green Pixel Value (Integer)
bhi	Maximum Blue Pixel Value (Integer)

Table 11. Continued

Character String	Description
rlo	Minimum Red Pixel Value (Integer)
glo	Minimum Green Pixel Value (Integer)
blo	Minimum Blue Pixel Value (Integer)

The final op-code for the tracking information is the ‘met’ descriptor. The method of feature detection is set by ‘met’. This can be set to: 0 for 3D Gaussian, 1 for 2D normalized Gaussian, 2 for color direction, or 3 for color range. This value is set independently for the foreground and background of each feature. Which ever method of feature detection gives the minimum error will be set as the method to use for that feature.

Figure 79 shows an example code snippet from a tracking file. This file contains data about tracking a single feature. The foreground and background information is only present for the one feature and one type of image processing, but it is sufficient for explaining the data file concept. The lines starting with ‘//’ are comments that have been placed in the data file. Any line that does not start with the proper control characters is ignored, so the comment indication could have been virtually anything. The double slashes were used for purposes of familiarity. In this example, both the foreground and the background have a ‘met’ of 0x00, which indicates that they are being tracked with the 3d Gaussian method. The *rme, *gme, etc. indicate the properties of the Gaussian distribution. The final entry for the foreground and the background is the Mahalanobis Distance for each distribution. For the foreground, a Mahalanobis Distance of 3.10 yielded the least error amongst the values tested. The number in parenthesis underneath

is the percent of error that was found at this Mahalanobis Distance. When this line is read by the position system, it is assumed to be a comment because it starts with a pair of spaces. Here a total error of 3.5% was found. This value is the sum of the errors from the desired pixels that were missed and the undesired pixels that were selected.

```
//Foreground 1 Statistical Model
1fmet 0x00
1frme 39.459727
1fgme 168.943223
1fbme 32.460770
1firr 0.038540
1figg 0.000769
1fibb 0.054827
1firg -0.002073
1firb -0.044613
1figb 0.001310
1f3dd 3.10
  (3.4857)

//Background 1 Statistical Model
1bmet 0x00
1brme 39.605433
1bgme 40.104653
1bbme 79.281011
1birr 0.051683
1bigg 0.049478
1bibb 0.005566
1birg -0.047995
1birb -0.003664
1bigb -0.001147
1b3dd 3.90
  (9.8212)
```

Figure 79: Sample Tracking Data File Automatically Generated from Training Data.

Finding Vehicle Features

Once the program has been initialized, the main loop starts. The main loop cycles through all of the equations specified by the user for each frame sequentially. The algorithm to locate and tag the vehicle features is fairly quick with respect to most of the initialization functions. The typical video stream runs anywhere from 12 to 30 frames per second so it is essential that this function be streamlined. Quantity was chosen over quality as the design paradigm for this portion of the program. The logic behind this decision is simple: if a vehicle is being tracked at a rate of 20 frames per second and the vehicle is not found for a frame then the position is only unknown for 5ms. A more comprehensive search of the image may find the vehicle in every frame, but can only search 1 frame per second. With this approach the vehicle's position is not known for a whole second between frame captures. In addition to a shorter latency period, the faster method also has the advantage of being

able to process more video streams simultaneously, further diminishing the chances of a failure.

Searching image

The feature detection stage has two major components: the image searching phase and the blob detection phase. The image searching phase is where all of the optimization is possible. The purpose of the image search is to find a single pixel that matches the foreground of either of the features. At this point there is not any concern given to the location of the feature or the likelihood of it being correct. This phase simply must find a matching pixel and call the blob detection algorithm for that pixel.

This function is fairly simple in nature, but it still must be run on a very large number of pixels. There is no way to speed up the inspection of individual pixels so the only optimization that can be done is to limit the number of pixels inspected. Two simple but effective methods are used for this. The first is to inspect every n^{th} pixel in the horizontal and the vertical. This effectively reduces the number of pixels scanned to $1/n^2$ of the original count. Using a skip rate of three or four will find most features in the image and dramatically improves the scan time. Features that are missed by skipping three or four pixels are usually too small to provide accurate information anyway. The second method is to alternate the features that are being looked for. It was found that searching for both features in a single pixel was

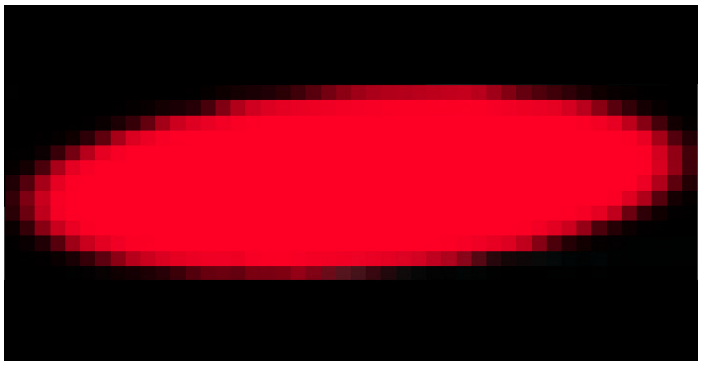

Figure 80: Typical Feature Seen by Camera

unnecessary. By alternating searches, the scan-time is again halved with minimal degradation to the search results.

A new concept that has not been included at the time of this paper, is a radial search pattern. With this method, the two features must be first found by the traditional method. Their pixel locations in the image are averaged to create a new starting point that is directly between them. On the next frame the search begins from the new starting point and concentric rings of pixels are searched expanding from this point. By doing this, the program is likely to find the feature quicker than by scanning top to bottom every time. If the features are not found for a few frames the program must start searching in the traditional pattern again. This search method will be added to the list of future work for this project.

Blob detection

Once a matching feature has been found by scanning the image, the more intensive blob detection algorithm can be called. The blob detector checks contiguous pixels for the feature's foreground color. If the foreground color is found then that pixel is marked as a foreground pixel. If the foreground color is not found then the accompanying background color is checked. If the background color is found then the pixel is marked as being a background pixel. If neither is found then the pixel is considered to be an 'outside' pixel and is marked appropriately, see Figure 81.

The blob detector is a recursive function that checks a single pixel's neighbors for feature colors. The blob detector searches the pixel above, to the right, below, and to the left of the current pixel (in that order). If the inspected pixel displays the feature background color or outside color, then the pixel is tagged appropriately. In the case of background and outside pixels, the search in that direction is terminated because there is

no reason to continue once the feature's border has been found. When the blob detector finds a foreground pixel, it calls itself on the new foreground pixel and continues the search from that pixel.

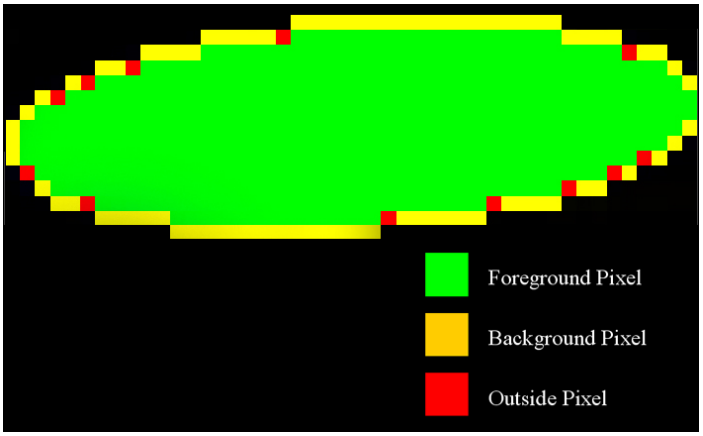

Figure 81: Blob Detection Result of Feature Shown in Figure 80.

Information about individual pixels is stored in an array that is $n \times m$ elements where n and m are the dimensions of the image. Each element of the array is a short integer and represents the information about a single pixel. The tag information is shown in Table 12.

Table 12: Blob Detector Pixel Tags

Character String	Description
0x00	Unchecked
0x01	Foreground Pixel
0x02	Background Pixel
0x03	Outside Pixel
0x04	Dark Pixel

An unexamined pixel starts out with a tag of 'unchecked'. Every time a neighbor pixel is inspected, it is first checked to see if the pixel has been looked at before. If a pixel neighbor is tagged as 'unchecked' inspection continues, otherwise the pixel is skipped. Doing this check first keeps the algorithm from having each pixel checked by every one of its neighbors and taking an excessive amount of time to run.

The optional ‘dark pixel’ tag is used to discard pixels that are close to black. When a pixel is very dark, the ratios between the color intensities can become very sensitive to noise. Noise accrued in video capture can change the color’s component values (red, green, and blue) by as much as 5 to 10 percent. So, if a color is very dark, this variance can make the same color appear to be shades of red, green, blue, yellow, etc. from pixel to pixel. This can confuse image processing algorithms, like color direction, that rely on the ratio of color components for detecting features. Tagging this pixel as dark will treat the pixel as a wildcard. When blob edges are searched, dark pixels will be ignored by the algorithm. For instance, if a blob has 100 background edge-pixels, 100 outside edge-pixels, and 200 dark edge-pixels, the ratios of pixel types will be ½ background and ½ outside. None of the dark pixels are taken into account. Of course, sometimes finding a feature that is black will be necessary, so the dark pixel search can be turned off if desired.

Returning blob information

The blob detection algorithm is called recursively, so depending on the size of the blob it may be called hundreds or even thousands of layers deep. This method is relatively quick and robust, but can require large amounts of system RAM. Compared to the image scan algorithm, the blob detection portion of the feature detection is very slow. While this algorithm has the built in property of determining the most likely blob to be a vehicle feature, it is advisable to limit the number of blobs the algorithm has to search. This is done by choosing appropriate tracking features and taking the time to create good training data for the statistical image processors. If adequate effort is placed in these tasks the position system should run quickly and accurately.

Determining Vehicle Location

The depth, or the number of recursions the blob detection function has been called, is tracked by a static variable in the function. The blob properties are tracked by a set of static variables as well. When the depth is zero, the function ends by compiling information about the blob and returning it to the calling function. This information includes the composition of the blob (number of foreground, background, outside pixels, etc.), the footprint of the blob in the image, the centroid, etc. This information can then be used to assess the value of the blob. If the blob is a decent size, and contains significantly more background pixels than outside pixels, it can be inferred that this blob is one of the vehicle tracking features. This ratio of background to outside pixels gives the likelihood that the foreground color is present on top of the background color. The more dissimilar the feature colors are to the rest of the image, the more meaningful this ratio becomes.

The blob's centroid location is returned in pixel coordinates as part of the blob information. This centroid is calculated by averaging the image coordinate locations of each of the foreground pixels in the blob. The result is guaranteed to be the centroid of the blob. If the tracking features are symmetrical across two perpendicular axes (i.e. a square or circle) a secondary centroid can be calculated to further verify the feature. It can be shown that the centroid for one of these types of features will lie at the midpoint of the high and low pixel values in each direction, see Figure 82.

The red dimensions show the known mathematic centroid of the shape. The quantity *H* indicates the horizontal dimension of the feature and *V* the vertical. The blue dimensions represent the image dimensions of the object. *MaxV* is the maximum vertical image location occupied by the feature and *MinV* is the minimum. *MaxH* and *MinH* are

the horizontal equivalents. It can be seen that the half-way points between these quantities represents the feature centroid.

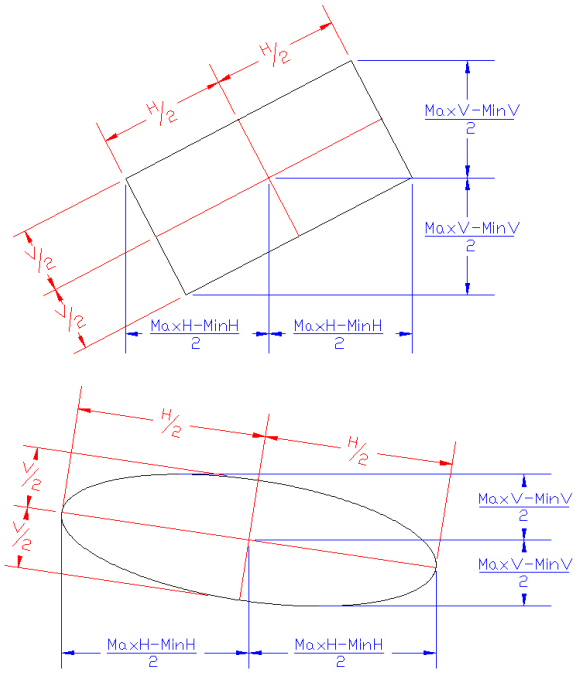

Figure 82: Symmetric Blob Centroid Calculation

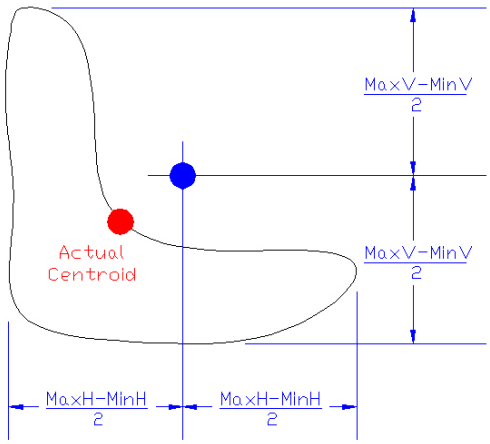

Figure 83: Non-Symmetric Blob Centroid Calculation

This technique only works for features that are symmetrical and can be used as a secondary check for blobs. If a blob is unsymmetrical, its centroid will not be centralized, see Figure 83. By comparing the centroids calculated by the pixel averaging and the half-way methods, it is possible to quickly tell if the blob is symmetric. This feature can be used to help eliminate invalid blobs found during feature detection.

Filtering Multiple Streams

For a visual position system to have any type of practical purpose, it must be able to handle input from multiple cameras. It is uncommon to have an environment that will allow for a single camera to see the entire area. Even if all the areas can be seen, position accuracy drops as the distance increases. To account for this, several video streams can

be processed simultaneously and their information integrated to determine the vehicle's position more accurately.

Unfortunately, the data from each camera cannot simply be averaged to get a better position value. One camera may have a clear, close-up view of the vehicle while another camera may see the vehicle from a distance. Obviously, the close-up camera should have the more accurate information, so a filtering system is necessary to appropriately weight position values with respect to their accuracy when averaging them. Each lookup table contains pixel-by-pixel error information that tells the maximum possible discrepancy for a pixel's real world location. The first stage of stream filtering takes just this information into account. The location of the feature can be found in world coordinates by the following equations:

$$X = \sum_{i=1}^{Nc} W_i \cdot x_i$$
$$Y = \sum_{i=1}^{Nc} W_i \cdot y_i \qquad (115)$$
$$Z = \sum_{i=1}^{Nc} W_i \cdot z_i$$

X, Y, and Z represent the location of the feature in the WCS when there are Nc cameras present. The variables x_i, y_i, and z_i are the xyz location of the feature as seen by the i^{th} camera. The variable W_i is the weighted averaging factor for the i^{th} camera. W_i is given by:

$$W_i = 1 - \frac{e_i}{\sum_{i=1}^{Nc} e_i} \qquad (116)$$

Where e_i is the max error associated with the pixel containing the feature for the i^{th} camera. X, Y, and Z now give the location of a feature in space with emphasis on streams with lower associated errors.

When the camera placement and functionality are assumed to be perfect, equations (115) and (116) are sufficient for modeling multiple cameras. In real life, this scenario of perfection is unrealistic. Some cameras may not work as well as others, giving images that are flawed and unreliable. Other cameras may not be measured off correctly, leaving residual error in range calculations. Perhaps a camera in a certain location is more prone to get oil and dirt on the lens, making its video difficult to see. There are many situations that can lead to diminished reliability from a particular video stream. For this reason, a user-definable reliability factor, R_i, is added into the equation. The product of the two averages are taken and placed into an intermediate variable w_i:

$$w_i = \frac{e_i^{-1} \cdot R_i}{\sum_{i=1}^{Nc} e_i^{-1} \cdot \sum_{i=1}^{Nc} R_i} \tag{117}$$

w_i cannot be used on its own to weight the averages because all its components do not necessarily sum to 1. To normalize this weighting factor it must be divided by the sum of all the w's:

$$W_i = \frac{w_i}{\sum_{i=1}^{Nc} w_i} \tag{118}$$

W_i is the new weighting factor that now takes the camera's pixel errors along with the user defined camera reliability factor into account.

The reliability factor R_i, while arbitrary, should be treated like a confidence percentage. For example, imagine a set of cameras with the same pixel errors for a given feature. One camera has a user-defined reliability of 100 and the other a 50. This will cause the camera with R=100 to get twice the weighting of the camera with R=50 for determining vehicle position. With this method, the user also has the option of completely disabling a camera via the reliability factor. A factor of R=0 will prevent that camera from contributing any information to the average.

CHAPTER 6
RESULTS AND CONCLUSIONS

Software Testing

This software was tested on three different environments: one simulated and two experimental. Each case uses multiple cameras to track a vehicle through an environment. The tracking results are shown in an X-Y plot displaying the actual and calculated location of the vehicle. The actual and calculated positions are compared for several frames in each case and error is calculated. The causes of error in the test case is discussed at the end of each section.

Simulated Test Case: Hazardous Materials Warehouse

Test setup

The simulation test case is the most crucial to proving the functionality of this system. Because no external sensors were used with this research, the accuracy of the position system on experimental models can only be verified visually. This type of verification is subjective and inaccurate, so a digital model was created that could be measured and mapped precisely.

A graphical model of a forklift and a warehouse was created. This model was made to closely resemble the type of real-life situation that this research could be used in. The premise of the model is a hazardous materials warehouse in which a barrel of noxious chemical has sprung a leak. The forklift is deployed to autonomously to remove the barrel from the rack. The warehouse contains several cameras that are used to track

throughout its mission. The warehouse model is shown, along with the camera placements in Figure 84.

The warehouse and forklift dimensions have been carefully measured to thousandths of a foot in the model. The camera parameters are also known with this same type of accuracy. Mapping and determining camera information is done with this level of precision to ensure that the incurred errors only come from the algorithms, not from inaccurate measurements.

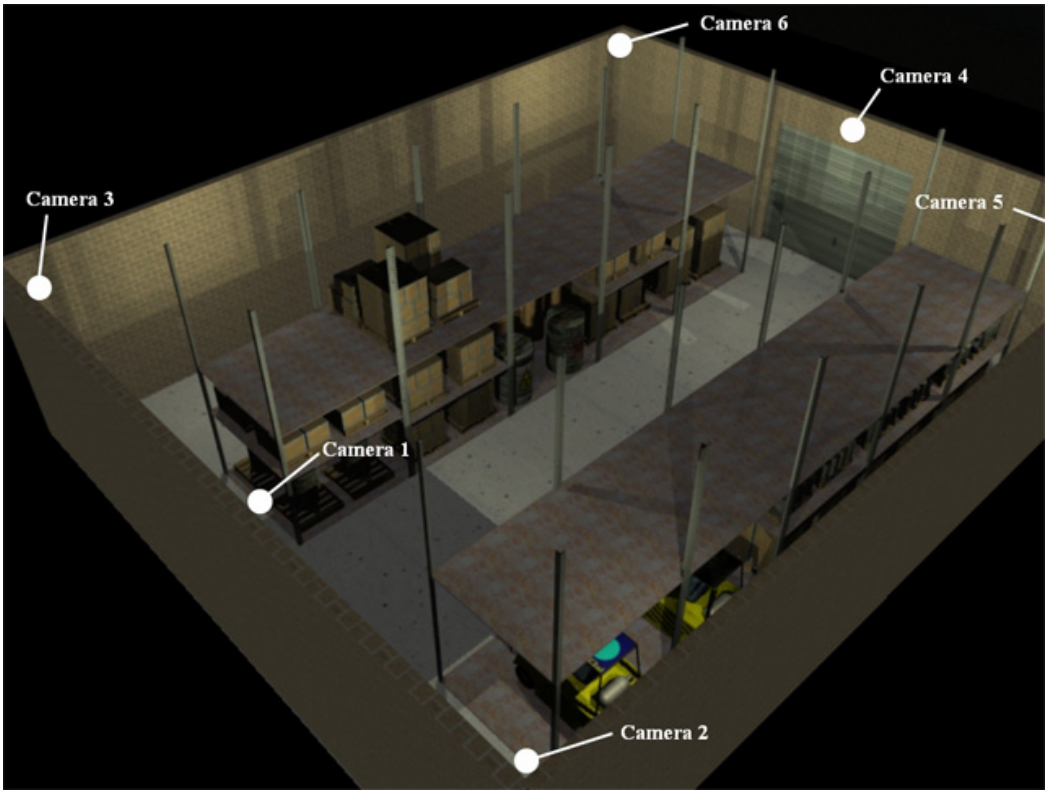

Figure 84: Overhead View of Simulated Warehouse with Camera Locations Shown

Only three of the camera views were rendered: Camera2, Camera3, and Camera6. These three vantage points are enough to show a proof of concept for the functionality of the system. In Figure 85 on page 167, there are nine images extracted from the original video file to show the path and progression of the forklift in the scene. When viewing the

figure in landscape format, the top row of images are all from camera 2, the middle row of images is from camera 3, and the bottom row of images are from camera 6.

The forklift travels in and out of the viewing volume for each camera throughout the span of the video, so each camera is only providing position information for the forklift for a short time. The full path of the forklift is shown in Figure 86 on page 168. The walls and storage racks of the warehouse are shown by thick black lines. Areas beyond these lines are considered to be 'out-of-bounds' by the software. The actual position of the forklift was measured on every tenth frame, and is shown by the red dots in the graph. The position of the forklift, as seen by the cameras, is determined in every frame so this data is represented by green, blue, and magenta solid lines for camera 2, camera 3, and camera 6, respectively.

After the graph of the full warehouse, there are separate graphs showing the data from each camera. Camera 2 is shown in Figure 87 on page 169, camera 3 is shown in Figure 88 on page 171, and camera 6 is shown in Figure 89 on page 174. These three graphs show zoomed in portions of the overall warehouse so that individual camera data can be seen more clearly. The color scheme stays consistent, but the camera data in these graphs appear with data points along with the solid lines so that the location at each frame can be inspected.

Finally, the error is checked between measured vehicle positions and the positions calculated by the software for each camera. This information is listed in tabular format for every ten frames. Camera 2 information is shown in Table 13 on 170, camera 3 information is shown in Table 14 on page 172, and camera 6 information is shown in Table 15 on page 175.

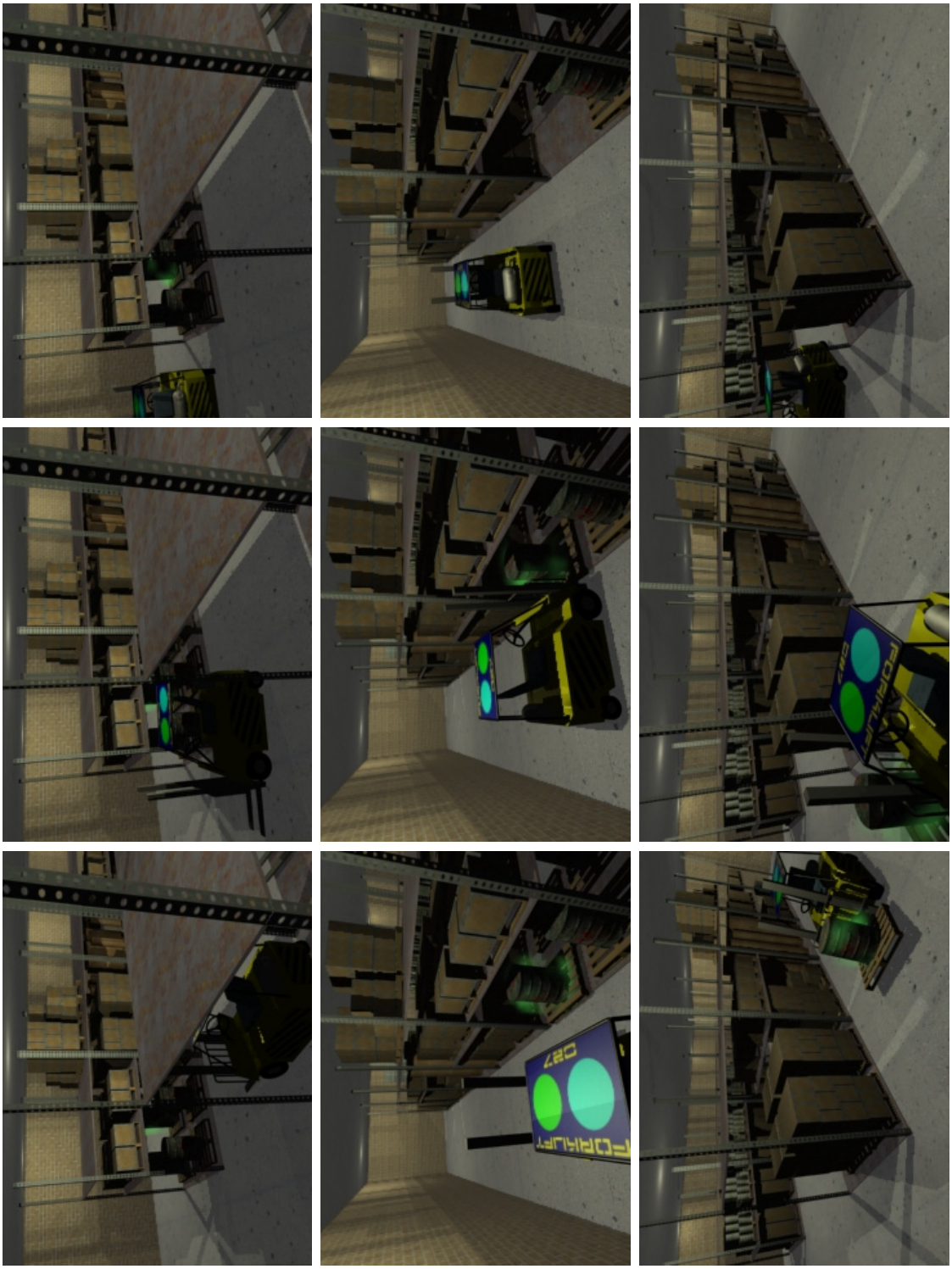

Figure 85: Frames Extracted from Hazardous Materials Warehouse Simulation. When Viewed in Landscape, the Images Represent: Camera 2 (Frame #016), Camera 2 (Frame #038), Camera 2 (Frame #076), Camera 3 (Frame #125), Camera 3 (Frame #203), Camera 3 (Frame #318), Camera 6 (Frame #376), Camera 6 (Frame #417), Camera 6 (Frame #472), Reading Left to Right and Top to Bottom.

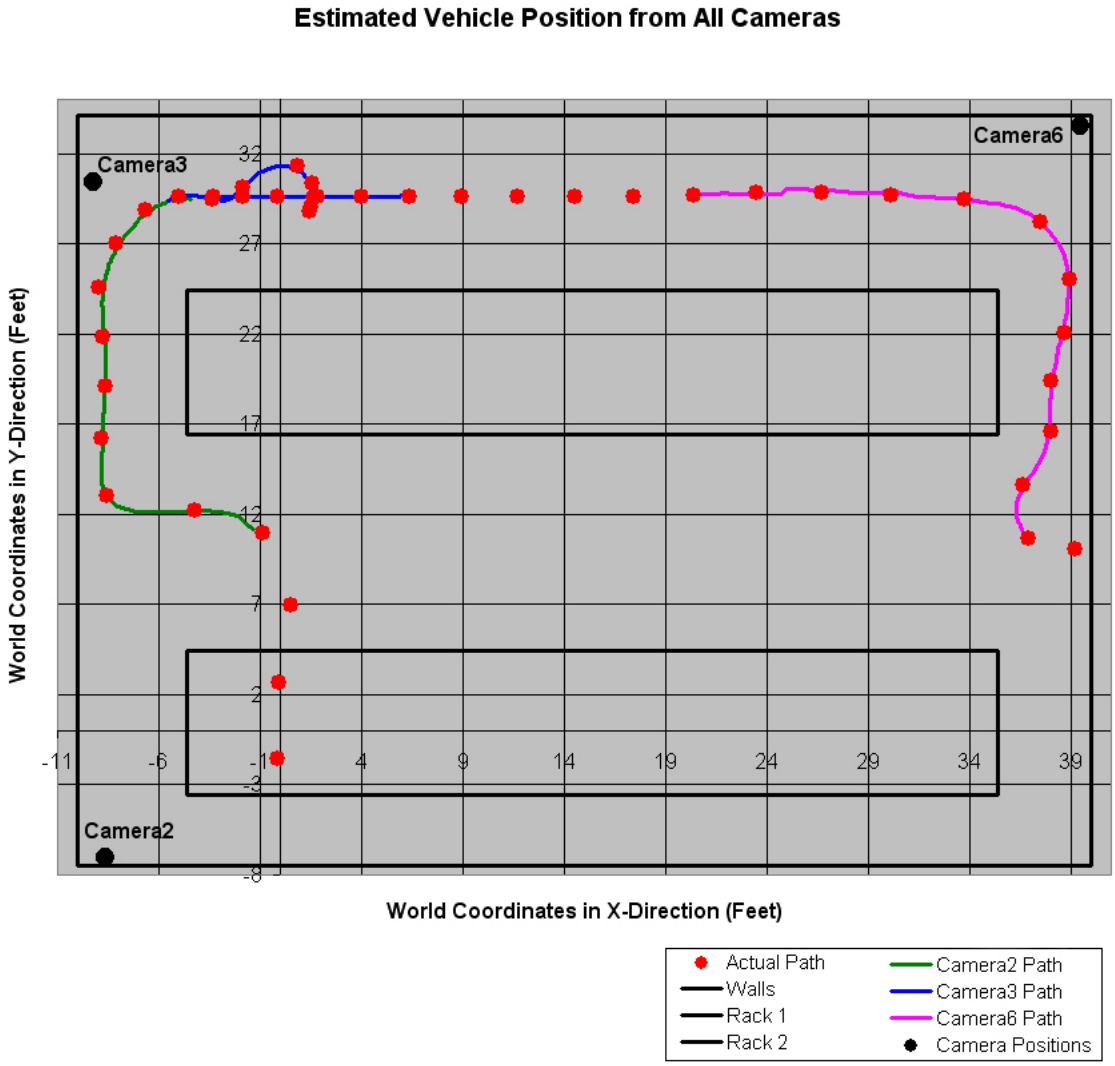

Figure 86: Estimated Forklift Path Determined by All Cameras

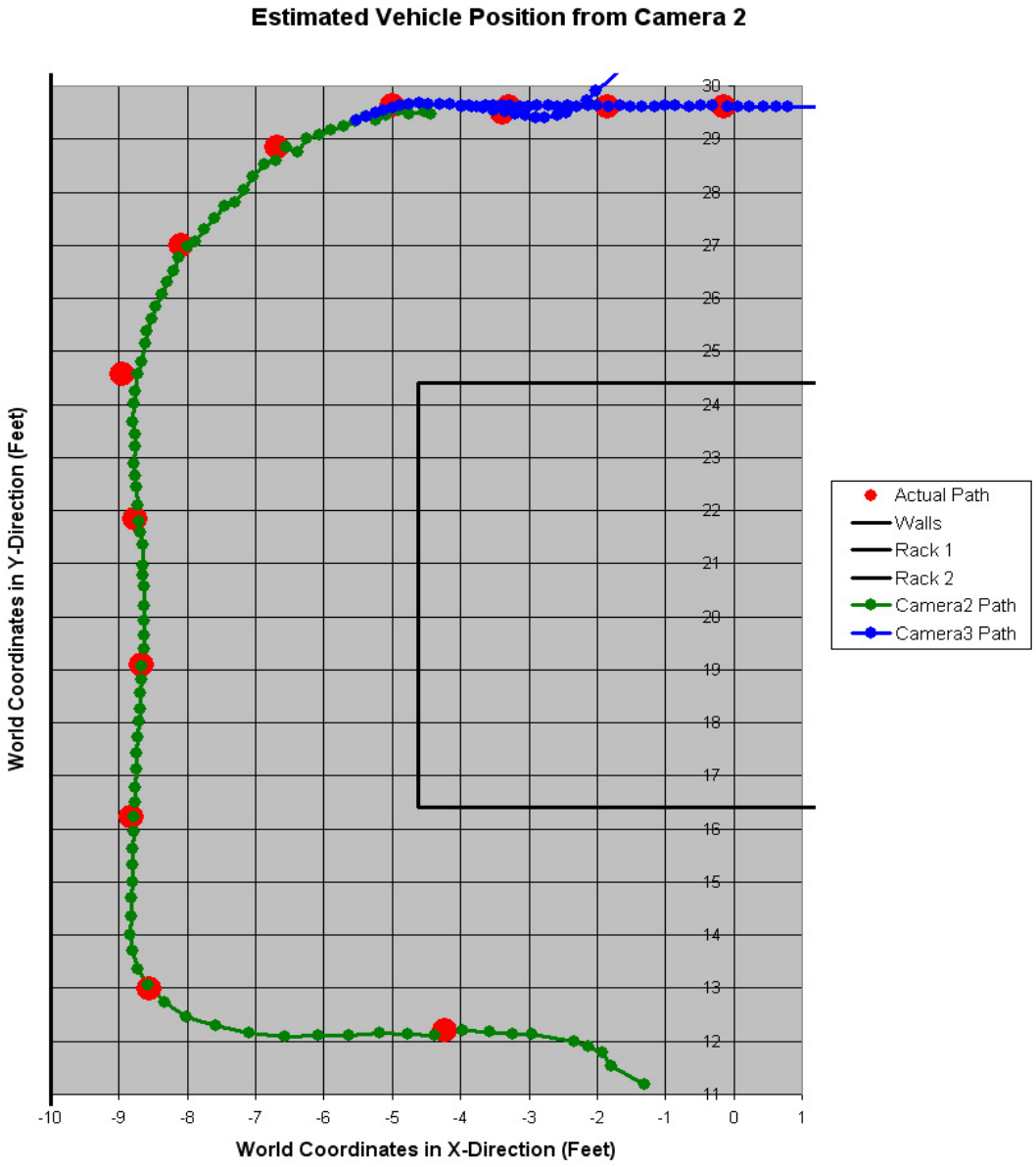

Figure 87: Estimated Forklift Path Determined by Camera 2

Table 13: Camera 2 Error Data

Frame No.	Actual X,Y	Measured X,Y	Error	Error Synopsis
0 - 30	---	---	---	Complete Obstruction of Tracking Features
40	-4.222 12.202	-4.378 12.109	0.182 (0.8%)	Partial Obstruction of Tracking Features
50	-8.558 12.985	-8.585 13.063	0.084 (0.4%)	---
60	-8.815 16.225	-8.776 16.221	0.039 (0.1%)	---
70	-8.662 19.094	-8.666 19.063	0.031 (0.1%)	---
80	-8.768 21.830	-8.705 21.786	0.076 (0.2%)	---
90	-8.950 24.569	-8.718 24.578	0.232 (0.7%)	Vehicle Partially Outside of Camera View
100	-8.092 26.998	-8.001 26.979	0.093 (0.3%)	---
110	-6.684 28.847	-6.560 28.837	0.124 (0.3%)	---
120	-5.005 29.630	-4.902 29.534	0.141 (0.4%)	---
130 -	---	---	---	Vehicle Completely Outside of Camera View

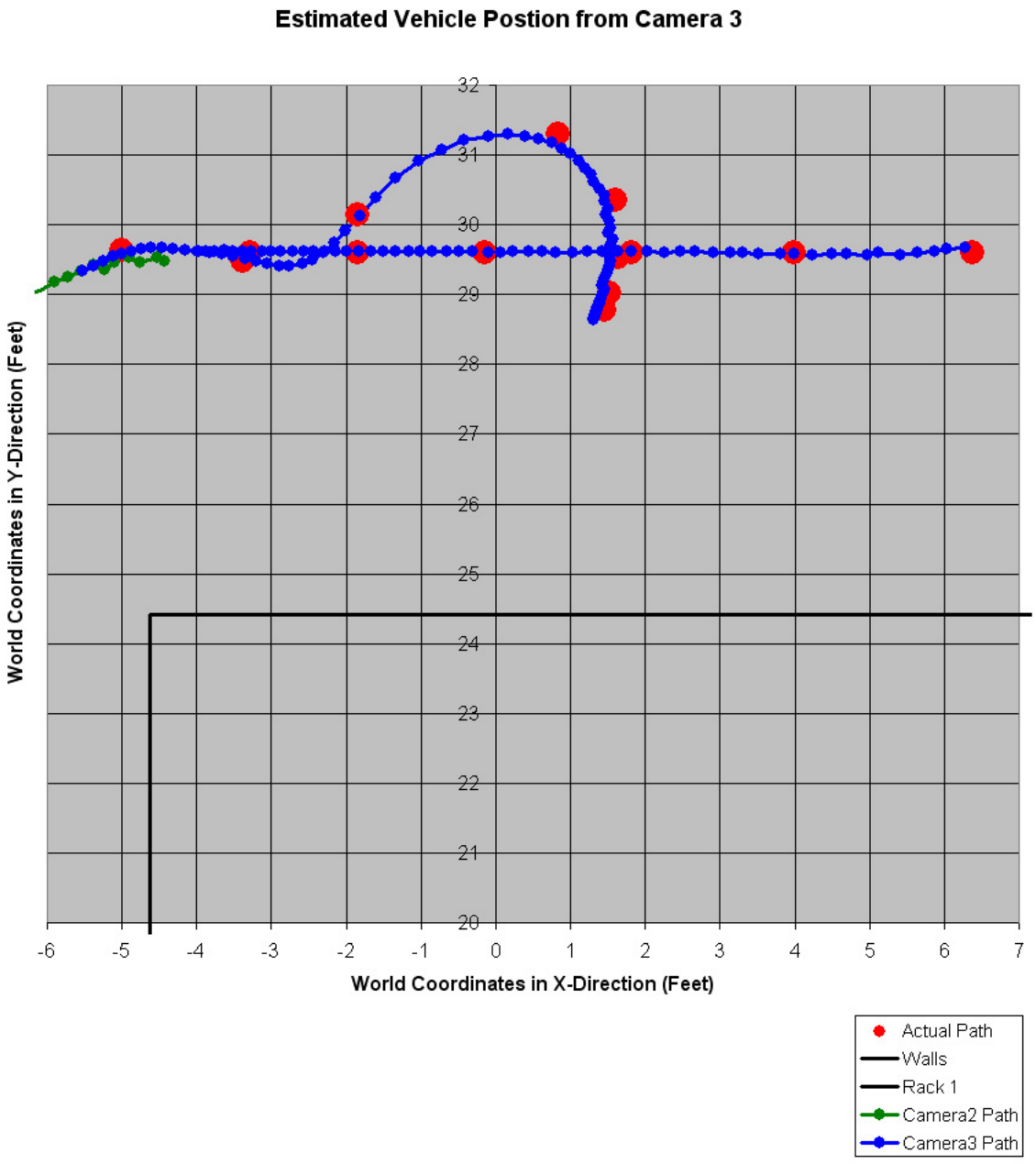

Figure 88: Estimated Forklift Path Determined by Camera 3.

Table 14: Camera 3 Error Data

Frame No.	Actual X,Y	Measured X,Y	Error	Error Synopsis
0 - 110	---	---	---	Vehicle Completely Outside of Camera View
120	-5.005 29.630	-4.878 29.621	0.127 (1.3%)	Vehicle Partially Outside of Camera View
130	-3.390 29.483	-3.349 29.509	0.049 (0.5%)	---
140	-1.845 30.148	-1.813 30.125	0.039 (0.3%)	---
150	0.830 31.291	0.748 31.174	0.143 (1.1%)	---
160	1.600 30.357	1.506 30.239	0.151 (1.1%)	---
170	1.638 29.531	1.521 29.405	0.172 (1.2%)	---
180	1.518 29.019	1.398 28.905	0.165 (1.2%)	---
190	1.452 28.775	1.319 28.670	0.169 (1.2%)	---
200	1.452 28.775	1.319 28.670	0.169 (1.2%)	---
210	1.515 29.016	1.398 28.905	0.161 (1.2%)	---
220	1.636 29.530	1.521 29.405	0.169 (1.2%)	---
230	1.601 30.353	1.506 30.239	0.149 (1.1%)	---
240	0.833 31.292	0.748 31.174	0.145 (1.1%)	---

Table 14. Continued

Frame No.	Actual X,Y	Measured X,Y	Error	Error Synopsis
250	-1.848 30.147	-1.813 30.125	0.041 (0.4%)	---
260	-3.389 29.483	-3.349 29.509	0.048 (0.5%)	---
270	-3.292 29.603	-3.267 29.623	0.032 (0.3%)	---
280	-1.855 29.603	-1.824 29.614	0.033 (0.3%)	---
290	-0.150 29.602	-0.093 29.605	0.057 (0.5%)	---
300	1.811 29.603	1.816 29.617	0.016 (0.1%)	---
310	3.994 29.598	3.990 29.590	0.009 (0.1%)	---
320	6.385 29.600	6.285 29.665	0.119 (0.7%)	---
330 - 430	---	---	---	Inadequate Tracking Feature Classification
440 -	---	---	---	Vehicle Completely Outside of Camera View

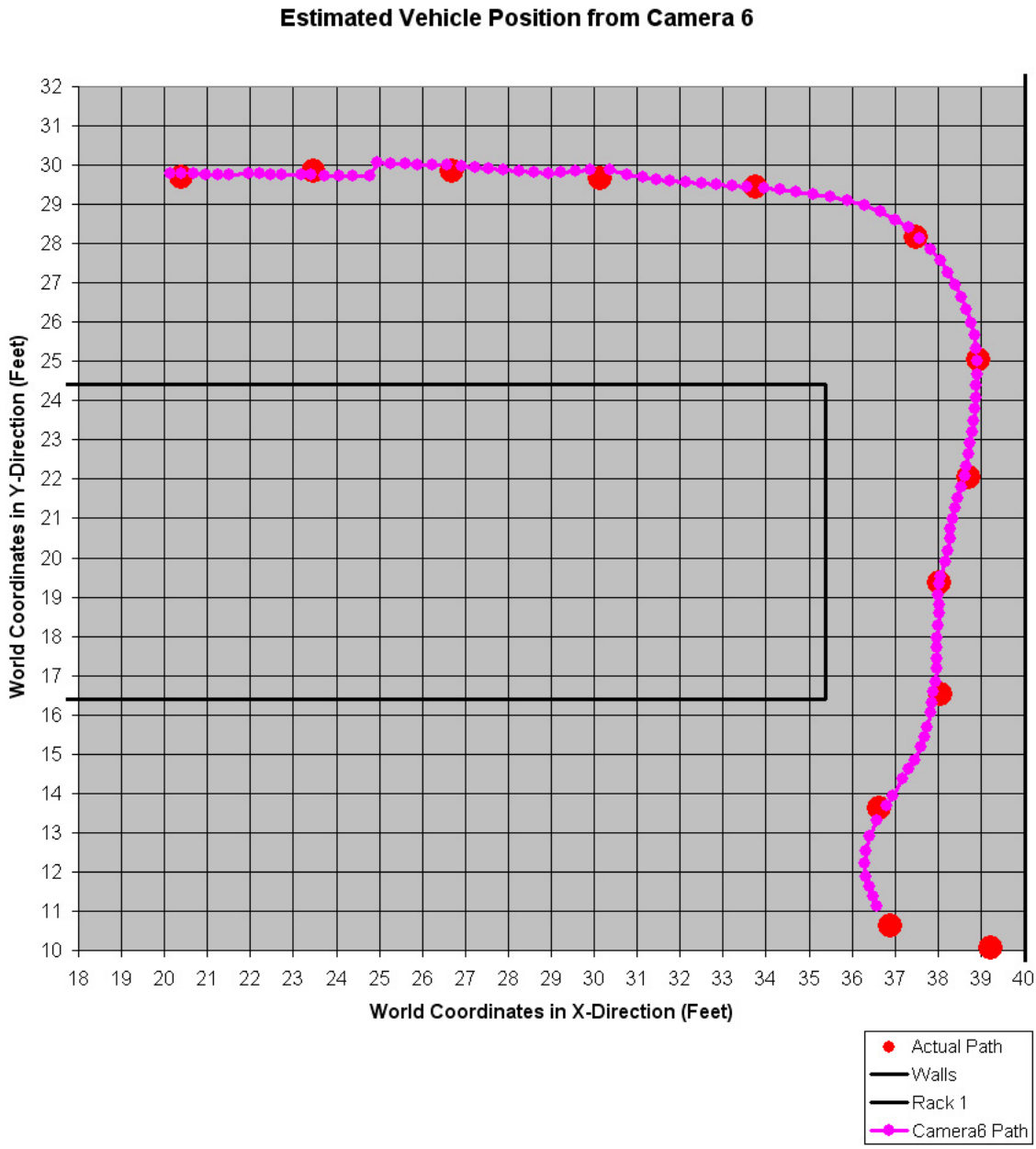

Figure 89: Estimated Forklift Path Determined by Camera 6

Table 15: Camera 6 Error Data

Frame No.	Actual X,Y	Measured X,Y	Error (%Error)	Error Synopsis
0 - 340	---	---	---	Vehicle Completely Outside of Camera View
350 - 360	---	---	---	Inadequate Tracking Feature Classification
370	20.406 29.706	20.682 29.777	0.285 (0.6%)	Partial Obstruction of Tracking Features
380	23.471 29.835	23.737 29.737	0.283 (0.6%)	Partial Obstruction of Tracking Features
390	26.689 29.854	26.911 29.959	0.246 (0.6%)	---
400	30.127 29.648	30.355 29.874	0.321 (0.8%)	---
410	33.768 29.445	33.945 29.424	0.179 (0.5%)	---
420	37.501 28.175	37.591 28.135	0.099 (0.3%)	---
430	38.945 25.036	38.912 25.013	0.040 (0.1%)	---
440	38.721 22.060	38.620 22.079	0.103 (0.3%)	---
450	38.035 19.373	38.029 19.330	0.043 (0.1%)	---
460	38.050 16.536	37.904 16.573	0.151 (0.4%)	---
470	36.640 13.624	36.798 13.687	0.170 (0.4%)	---
480 -	---	---	---	Vehicle is Outside of Camera View

Test results

The results from the simulated test case were better than expected. When the forklift was in clear view of the camera, the resulting accuracy of the system was excellent. The error in vehicle position stayed in the range of a couple of inches, typically yielding less than a 1% error. Even when tracking features were partially obstructed, the maximum measured error was only 1.3%.

The greatest problem with the test was the inability of the feature extraction algorithms to consistently find the tracking features. A portion of the test did not yield tracking data because the software could not distinguish the tracking features from the rest of the image. Specific issues that had to be dealt with in this test are discussed below.

Vehicle completely outside of camera view

This error is present at one point or another for each of the cameras. This type of error occurs when the vehicle leaves the viewing area of the camera. This is a common problem and is typically unavoidable. For most situations, the vehicle will not be in view of every camera at all times. As long as, at least one camera can see the vehicle, position data will still be calculated.

This becomes a problem if there are locations in the environment where no cameras can see the vehicle. Obviously, if the vehicle cannot be seen, it cannot be tracked. This problem can be avoided if the cameras are placed in locations that allow for complete coverage of the tracking area.

Vehicle partially outside of camera view

This error occurs when the vehicle enters and leaves a camera's viewing area. When the vehicle is partially in view of the camera the features can still be tracked, but the location of the centroids will be shifted. Figure 90 shows frame 120 of the camera 3 video feed. The rear feature is partially outside of the view, and the centroid of the feature is noticeably displaced. This type of error only caused a 1.3% discrepancy in the calculated location of the vehicle. Partial view errors are typically unavoidable, but the resulting degradation in position information is minimal.

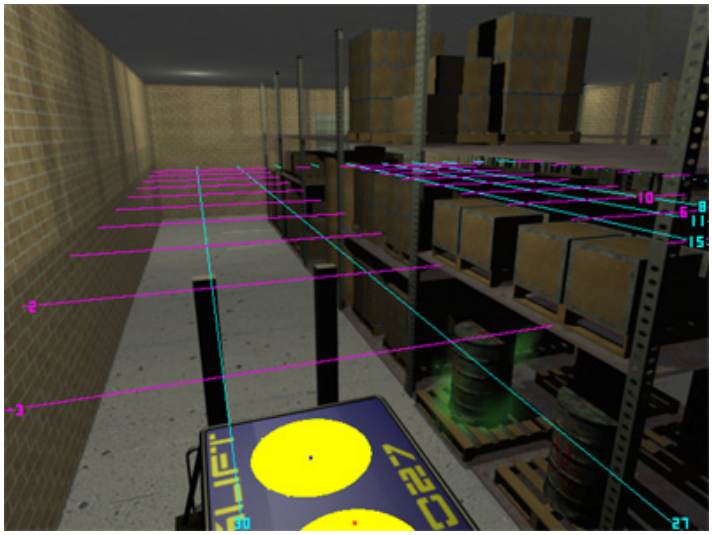

Figure 90: Vehicle Partially Outside of Camera View.

Inadequate tracking feature classification

Tracking feature classification is consistently the portion of the program that creates the most error. Both simulated and experimental models are subject to a large range of lighting conditions. As lighting conditions change, so do the colors of the features. This makes it quite difficult to classify the tracking features in a way that will both allow for the detection of the vehicle and avoid detection of other objects in the scene.

Figure 91 shows frame 330 of the warehouse simulation from camera 3. At this point in the video, the forklift is approximately 25 feet away from the camera. The tracking features at this distance are beginning to exceed the boundaries of the classifier. The position data is only calculated when both of the tracking features can be found, so at this point the vehicle location is unknown. This type of error is troublesome and occurs relatively frequently. A more powerful feature classifier may be required to eliminate this type of error.

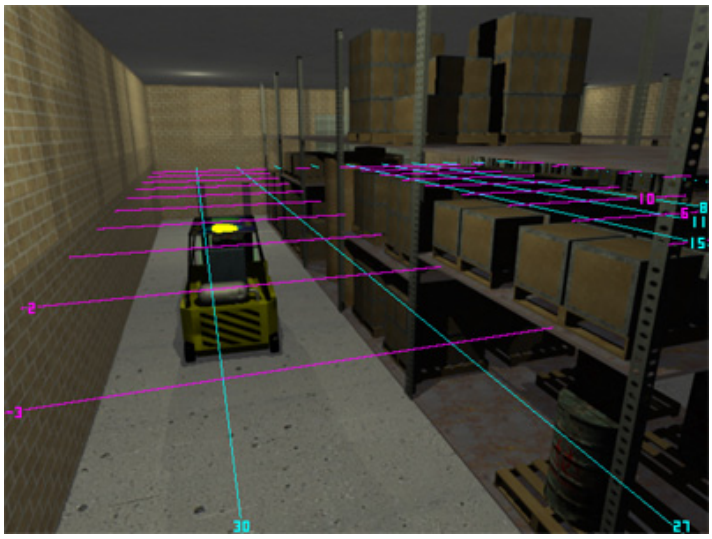

Figure 91: Vehicle Partially Outside of Camera View.

Partial obstruction of tracking features

For this simulation, cameras were intentionally placed in positions that would cause partial and total obstruction of the tracking features. This was done so that the position system's ability to deal with these types of situations could be assessed. Figure 92 shows frame 370 from camera 6. In this image, both of the vehicle tracking

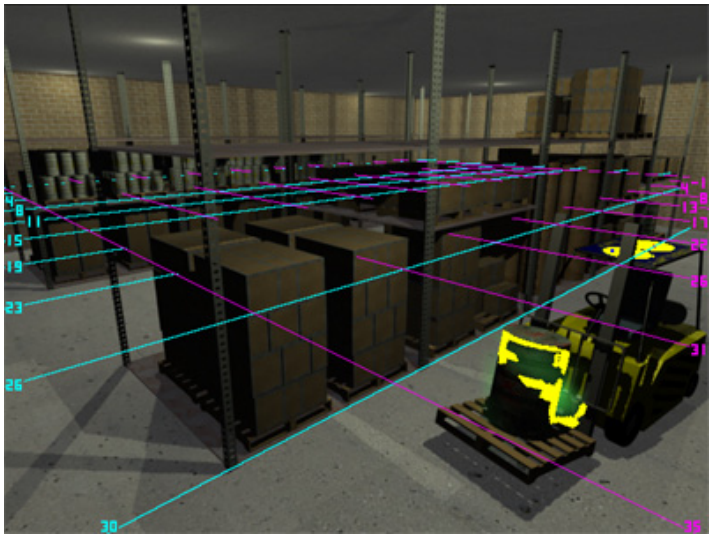

Figure 92: Partially Obstructed Vehicle Tracking Features.

features are being partially obstructed by the upright rails on the forklift. Even though the centroids of the tracking features appear to be significantly shifted, the resulting error in this frame is only 0.6%. Partial tracking feature obstruction errors cause minimal errors.

Complete obstruction of tracking features

Depending on the setup of the environment, it may be possible to have an object completely obstruct the tracking features. When one or both of the features are completely obstructed, no position information is available. Figure 93 shows frame 30 from camera 2. In this image the rear tracking feature is completely hidden by the storage rack. At this instance, no position information is known about the vehicle. This situation can usually be avoided by carefully arranging the cameras.

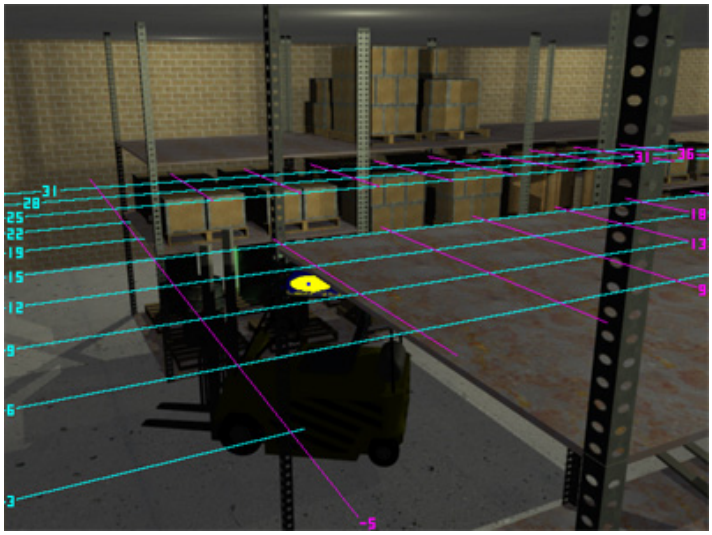

Figure 93: Completely Obstructed Vehicle Tracking Feature.

Experimental Test Case #1: Miniature Desktop Rover Vehicle

Test setup

The first of the three experimental test cases is of a miniature remote controlled desktop rover vehicle. The environment for this test case is a large poster board platform with world coordinate gridlines drawn on to the surface. The gridlines are used to visually inspect the location of the vehicle. Since the vehicle has no onboard position system, these gridlines are the only way to determine the actual position of the vehicle.

Hardware used in this experiment remained constant for all of the test cases. The hardware is listed in Table 16.

Table 16: Video Capture Hardware List

Item	Description
Panasonic PV-L559D Camcorder 	The PV-L559D is a typical home camcorder with some of the following properties: - 26x Optical Zoom - Auto-Iris - Auto-Focus - Digital Image Stabilization
Sony XC-711 Industrial Video Camera 	The XC-711 is typically used in industrial and security applications. This camera has: - Manual-Focus - Manual-Iris - Fixed Lens (No Zoom)
X-10 Wireless Video Camera 	The X-10 video camera is an inexpensive wireless security camera. This camera has: - Auto-Focus - Auto-Iris - No Zoom Capabilities
Dazzle DVC-100 Digital Video Capture Device 	The DVC-100 is a USB video capture device that can encode video from composite or S-Video sources. The DVC-100 has a built in MPEG encoder that converts video to MPEG2 compression real-time.

Video of the test was taken by two different cameras: an older model Panasonic camcorder and an industrial Sony XC711 CCD video camera. The two cameras are very different in nature and provide good discrepancy between views. The camcorder has the standard features: auto-focus, auto-iris, and zoom capabilities. The CCD camera, on the other hand, has manual-focus, manual-iris, and a fixed lens. These two cameras produce very different video streams and are useful in determining the robustness of the software across different video capture conditions.

Two tests on this system were done: the first was outdoors and the second was indoors. The outdoors test had to be cut short because of rain, so the test was then setup indoors and re-run. The desktop rover vehicle and original test setup are shown in Figure 94. Since the outdoor data is incomplete, only the indoor test is discussed in this section.

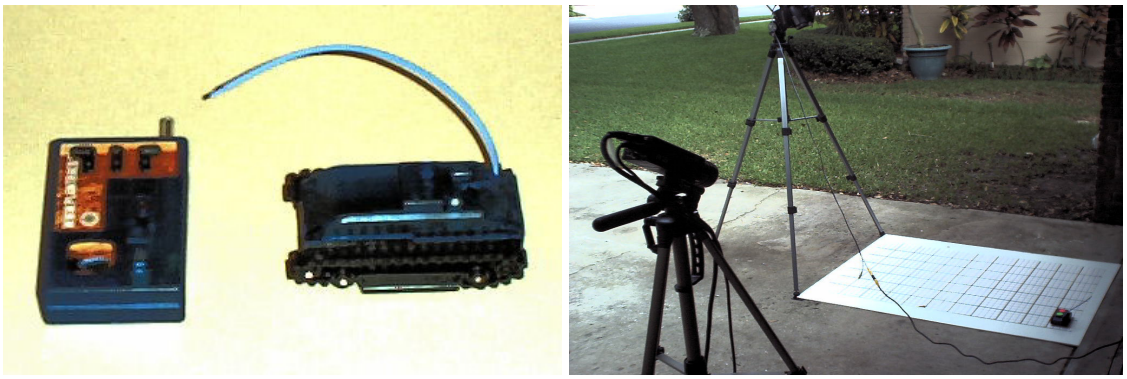

Figure 94: Experimental Test Case #1. Miniature Desktop Rover Vehicle (Left). Outdoor Test Setup (Right).

For the indoor test, the vehicle was driven around on the grid platform and stopped occasionally so that vehicle position could be inspected. The position system software was run on the video from both of the cameras to create a map of the vehicle's position throughout the test.

The accuracy of the position system for this case was tested in a couple of ways. The first, and simplest, test entails aligning the feature tracking plane with the ground.

When coordinate gridlines are added by the position system software, the proximity to features in the image can be checked. For this case the calculated gridlines are defined so that they lie on top of the lines drawn on the poster board, see Figure 95.

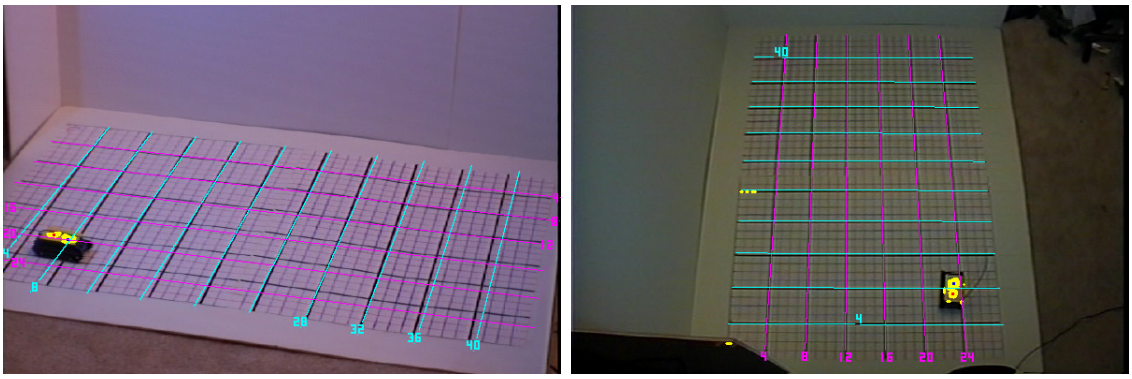

Figure 95: Gridline Accuracy Tests. CCD Camera Test (Left). Camcorder Test (Right).

This test on the camcorder yields excellent results. The gridlines are almost exactly on top of the actual measured gridlines. Also, the gridlines end at the exact border of the tracking area. Therefore, the camcorder positioning is very accurate. On the other hand, the CCD camera results are not great. There are discrepancies on either side of the image where the calculated gridlines do not meet up with the lines on the poster board. From the data in these two tests, it can be concluded that emphasis should be put on the camcorder data whenever possible.

The image quality obtained from each camera is a different story. The CCD camera has a much more vibrant picture, allowing for exceptional feature extraction. In this category, the camcorder falls short. The color contrast obtained from the camcorder is minimal resulting in less separation between colors. This makes finding vehicle features more difficult and more prone to errors on this video.

The result of these vastly different cameras can be seen in the vehicle position graph. The camcorder data, shown by the blue data points, depicts the more accurate vehicle path. While the position from the camcorder video is more accurate, there are

many points in the video that the software cannot find the tracking features. In other parts of the video, incorrect features are found.

The CCD camera data, shown by the magenta data points, has error that increases with the distance from the camera, but very few feature detection errors. The combination of the data from these two cameras gives a consistent vehicle location path that exhibits some minor errors, but is better than either video stream individually.

Images from this test are shown in Figure 96 on page 184. These images show the view from both cameras for three different instances during the test. The left column shows the view from the CCD camera for each of these instances, and the right column shows the view from the camcorder for each of these instances.

Position data was taken from both cameras throughout the test and combined to determine the vehicle's path in Figure 97 on page 185. Data from the camcorder is depicted by the blue diamonds and data from the CCD camera is shown by the magenta squares. The vehicle location, as determined from both sets, is shown as a yellow line. Nine frames were visually inspected for position using the gridlines drawn on the poster board. These data points, shown as red dots in the graph, are used to calculate the position system error. The results from these tests are shown in Table 17 on page 186. The camera locations are shown in Figure 98 on page 186.

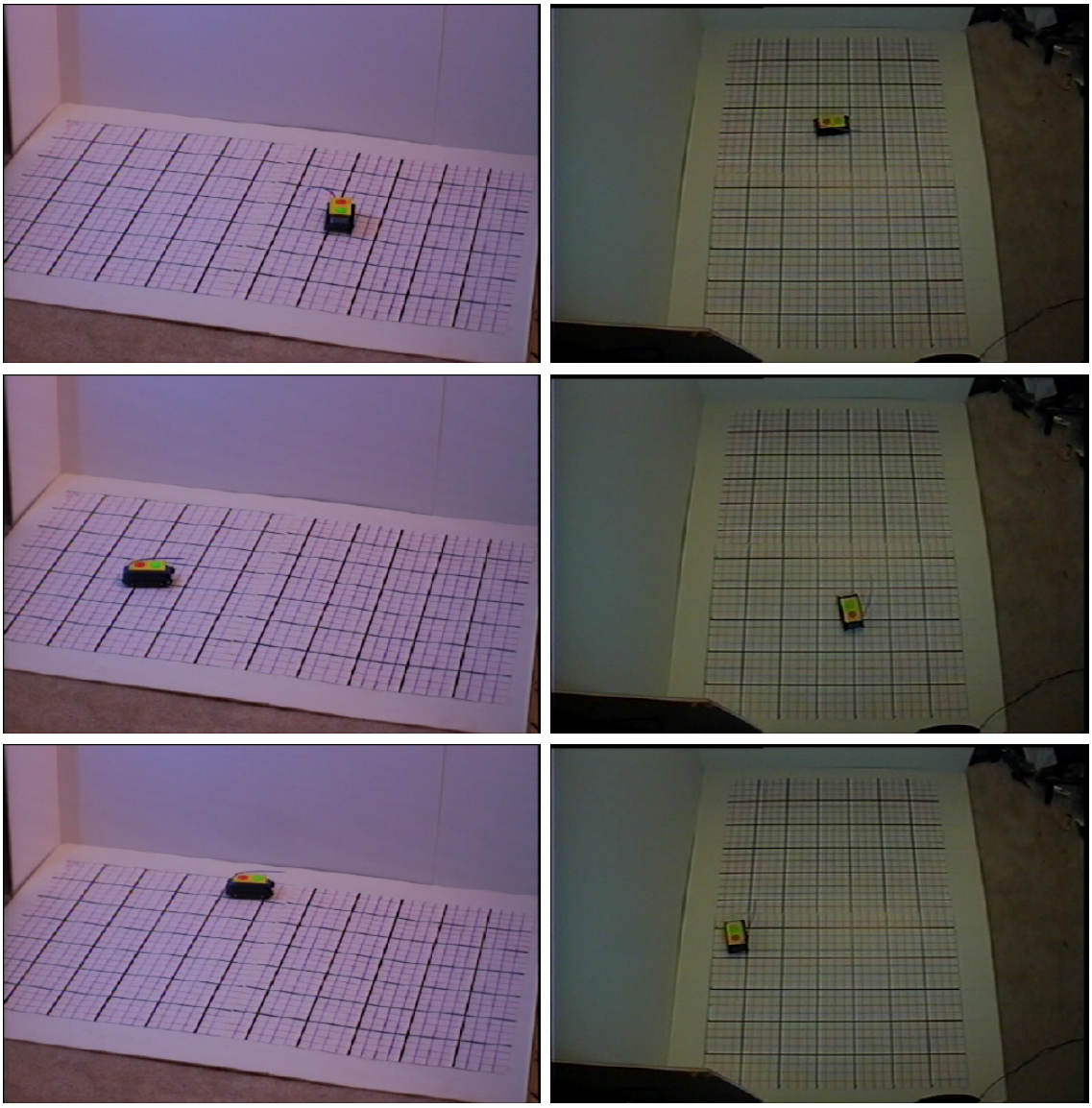

Figure 96: Frames Extracted from Miniature Desktop Rover Video. Each Row Represents a Single Frame in the Video Shown by Both Cameras. Frames Taken by the CCD Camera are Shown in the Left Column. Frames Taken by the Camcorder are Shown in the Right Column.

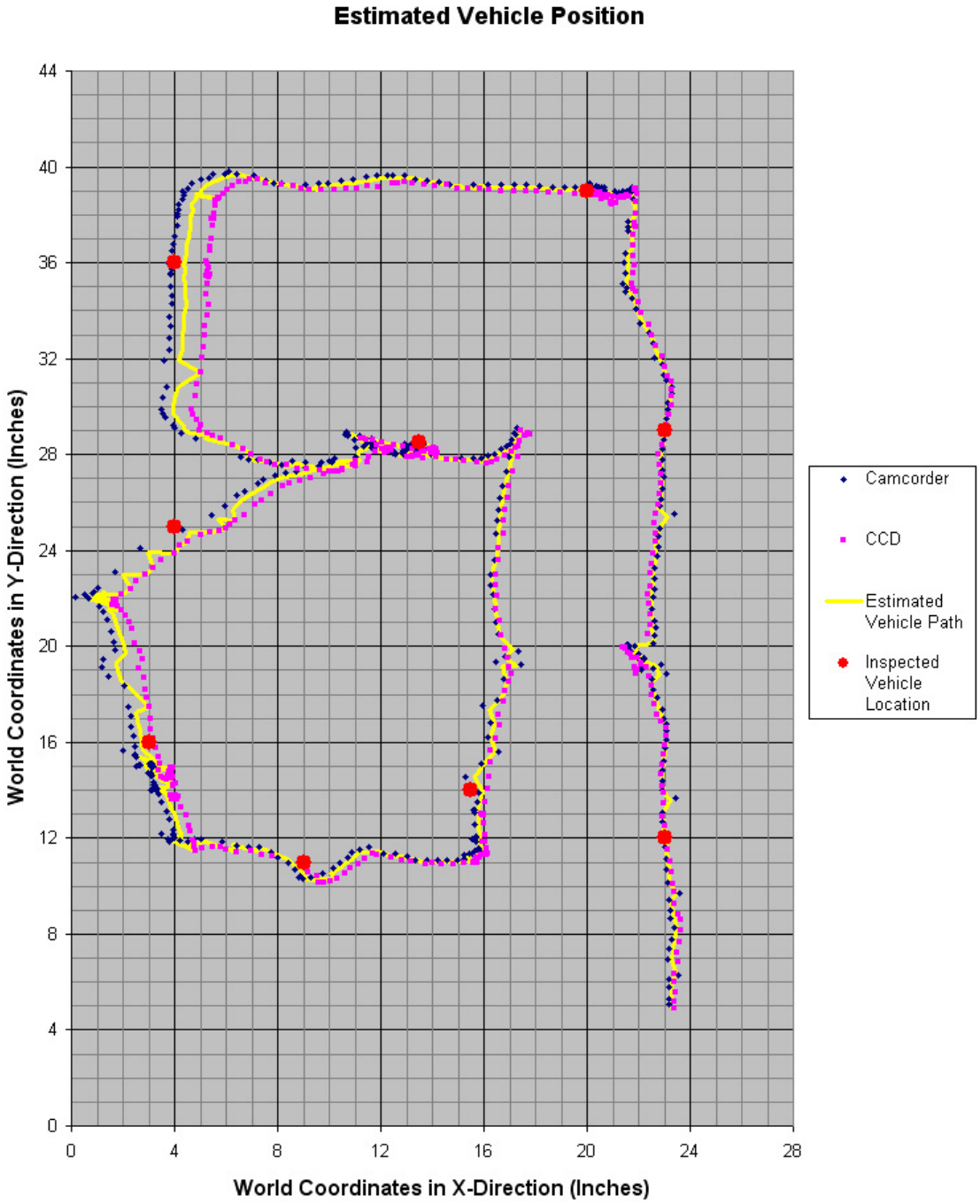

Figure 97: Calculated Vehicle Position and Estimated Path of Mini Rover.

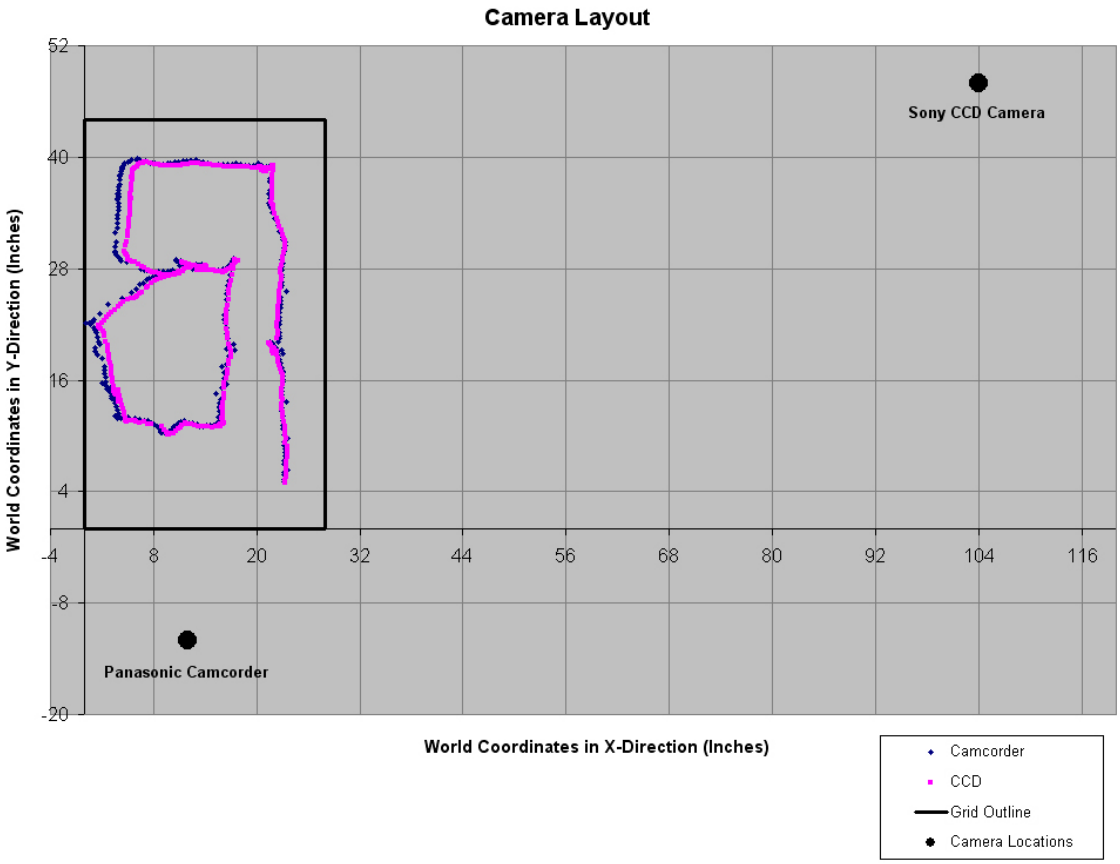

Figure 98: Location of Cameras with Respect to Tracking Area.

Table 17: Desktop Rover Tracking Error

Frame No.	Examined X,Y	Measured X,Y	Error (%Error)	Error Synopsis
15	23.0 12.0	23.083 11.162	0.842 (1.2%)	---
73	23.0 29.0	22.991 28.626	0.374 (0.5%)	---
170	20.0 39.0	20.405 39.057	0.409 (0.5%)	---
250	4.0 36.0	3.838 35.484	0.541 (0.7%)	---
319	13.5 28.5	13.654 28.234	0.308 (0.4%)	---
406	15.5 14.0	15.664 13.044	0.970 (1.4%)	Results Based Solely on CCD Camera Data

Table 17. Continued

Frame No.	Examined X,Y	Measured X,Y	Error (%Error)	Error Synopsis
460	9.0 11.0	8.892 10.434	0.576 (0.9%)	---
543	3.0 16.0	2.527 15.365	0.792 (1.1%)	---
579	4.0 25.0	3.977 23.892	1.108 (1.5%)	Results Based Solely on CCD Camera Data

Test results

The results from this test case were surprisingly good. For the nine frames that were tested against examined location, the maximum error was only 1.5%. It is likely that there are some frames with higher error rates, but the results of the system as a whole are quite good. Some of the same problems from the previous test were present again in this one, and some new types of problems arose with the induction of experimental video. Each of these problems is discussed in the following sections.

Inadequate tracking feature classification

Creating a classifier that could track the vehicle data decently without causing false hits was again a problem. The classifier used on the CCD camera's video worked very well, but the low contrast of the camcorder caused problems. In order to minimize the finding of false features, the classifier had to be narrowed down significantly. Because of this, there are many points in the camcorder video where the tracking features cannot be found.

Problems with compressed video data

A new, and unexpected, problem arose when the recorded video was converted to digital format. The frame grabber used to capture the video from the cameras, a Dazzle DVC 100, automatically interlaces and compresses video to MPEG2 format. These two processes significantly damage the video stream, making it especially difficult to extract color information.

Figure 99: Image Degradation from Interlacing and MPEG Compression.

Results from interlacing and MPEG2 compression are shown in Figure 99. This image shows a close-up view of the tracking features on the desktop rover vehicle. The most obvious problem with the image is the separation of the green feature due to interlacing. Interlacing only becomes noticeable with motion. The faster an object moves in the scene, the more effect interlacing will have on the image.

The change of pixel information due to MPEG2 compression is not as clearly visible as interlacing, but it is much more troublesome. When the capture device encodes video data into MPEG, the original colors in become scattered and blended. Even though this change is minute, the effect to the classifier is profound. These minor changes appear in the training data as excessive scatter in the color coordinates, making it difficult to separate the feature data from the non-feature data. The net result is an increase in the number of incorrect features found in the scene, making it more difficult to discern the correct tracking features.

Inaccurate camera model

The reliability and precision of the position data in this system relies greatly on the accuracy of the measurements of the cameras in the scene. While it is a trivial task to determine the x, y, and z coordinates of the camera, it is not as easy to accurately determine the rotation angles or fields-of-view for the camera. These values must be incrementally adjusted to get the camera model to properly fit the environment. For the camcorder this was not a problem, but the CCD camera was consistently difficult.

Experimental Test Case #2: Remote Control Truck

Test setup

The second experimental case was done with an inexpensive remote controlled truck, shown in Figure 100. The test environment for this vehicle was a tiled kitchen floor. This location was chosen because the tiles could be used to create a location grid in the environment to check the accuracy of the position system.

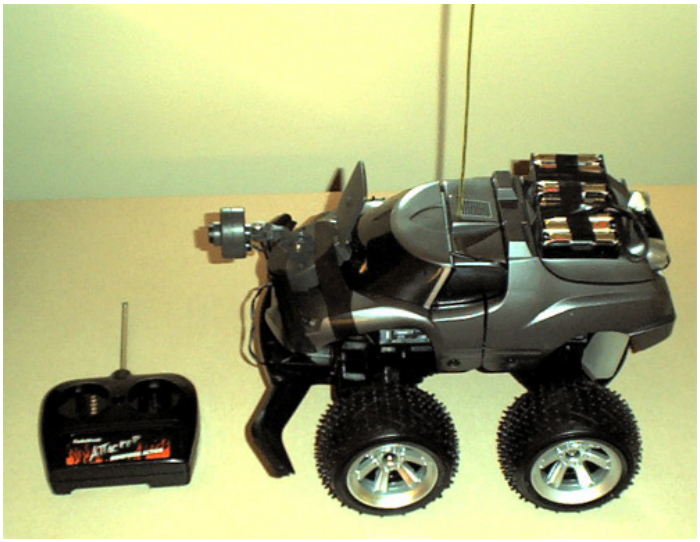

Figure 100: Vehicle Used in Experimental Test Case #2

The same two cameras from the previous test case were used: the Panasonic camcorder and the industrial Sony video camera. In this test case, the lighting conditions yielded similar image quality from both the CCD camera and the camcorder. Determining a good positioning data from the CCD camera was again difficult resulting in diminished reliability in the data from that camera. In addition to the two tracking cameras, a third camera was added onboard the vehicle.

This wireless security camera was used to show the environment from the vehicle's perspective throughout the test. This camera is shown in Figure 100 strapped to the hood of the vehicle.

Like in the previous test, the vehicle was driven around the environment while simultaneously being taped by the tracking cameras. Figure 101 on page 192 shows three sets of images extracted from the videos in this test. When the figure is viewed in landscape format, each row of images corresponds to a camera. The top row shows images extracted from the camcorder video, the middle row shows images extracted from the CCD camera video, and the bottom row shows images extracted from the onboard wireless camera. Each column represents the views shown by the three cameras at a single instance in the test.

The view from the camcorder is clear and in-focus for all visible areas. The camcorder displays video with good contrast in this environment, unlike in the previous test. The CCD camera displays the same image quality as from the last test, but this time focus is an issue. The tracking area is relatively deep in this test making it difficult to bring the entire image into focus. To account for this the camera was focused on the middle of the tracking area to minimize the overall blurring of the image. The result is an image that is slightly blurred at the near and far ends of the tracking area.

The final video, coming from the wireless onboard camera, displays very poor quality. This particular camera is fairly inexpensive, and therefore, does not have the image quality of the other cameras. To start with, this camera only captures 320 scanlines per frame, instead of the 752 captured by a standard NTSC camera. The camera does have an optical focus or iris, so it is very sensitive to environmental

conditions. This camera is designed for outdoor usage so the lighting in this test was inadequate for quality video. Fortunately, this video is not used for feature extraction so the quality is not an issue.

The vehicle position, as determined by the video cameras, is shown mapped in graphical format in Figure 102 on page 193. The camcorder tracking coordinates are shown in blue and the CCD camera tracking coordinates are shown in magenta. The estimated position of the vehicle is shown in red.

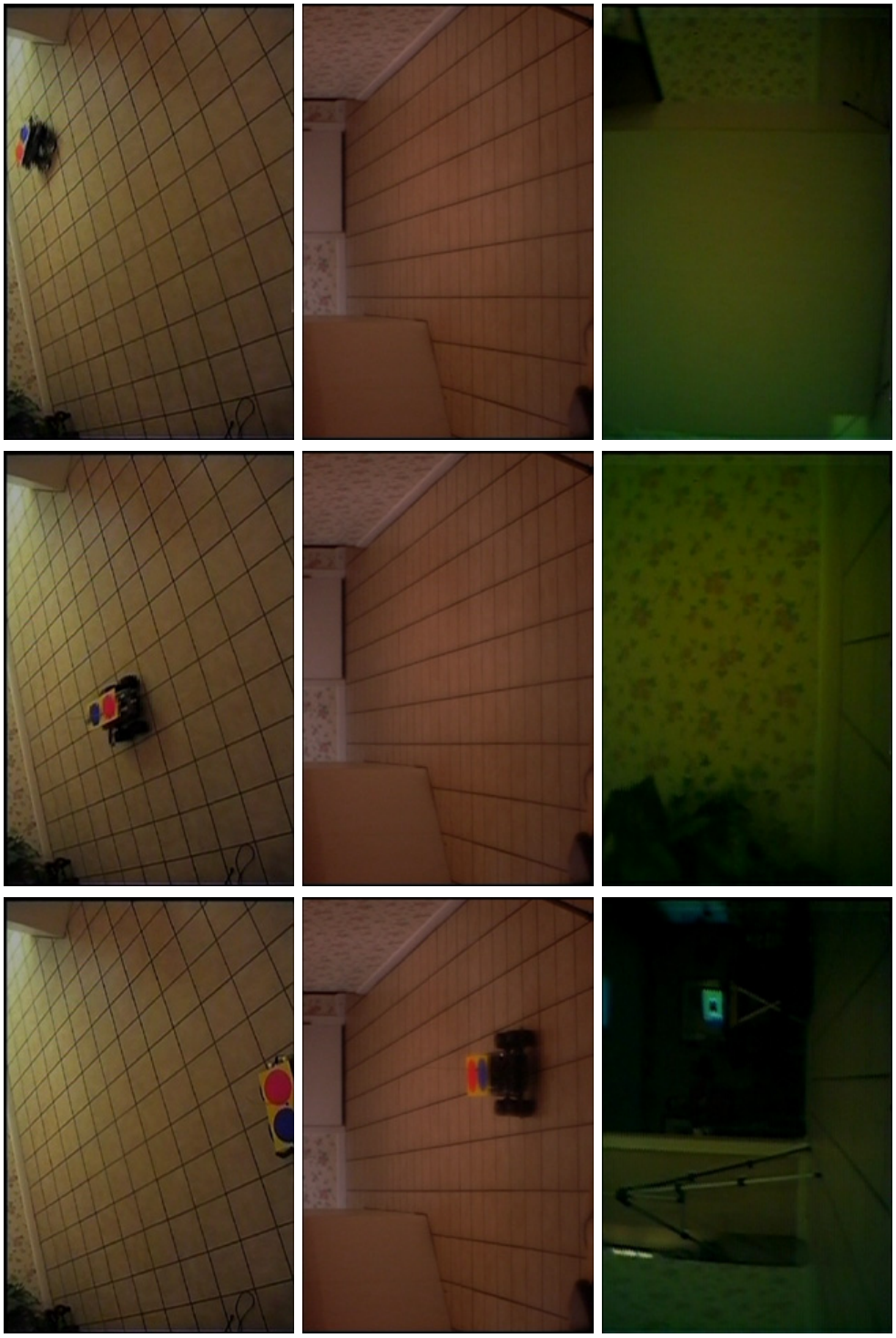

Figure 101: Frames Extracted from Remote Control Truck Video. When Viewed in Landscape, Images Taken from the Camcorder (Top Row). Images from the CCD Camera (Middle Row). Images from the Wireless Onboard Camera (Bottom Row). Each Column Shows a Single Instance Shown by All the Cameras.

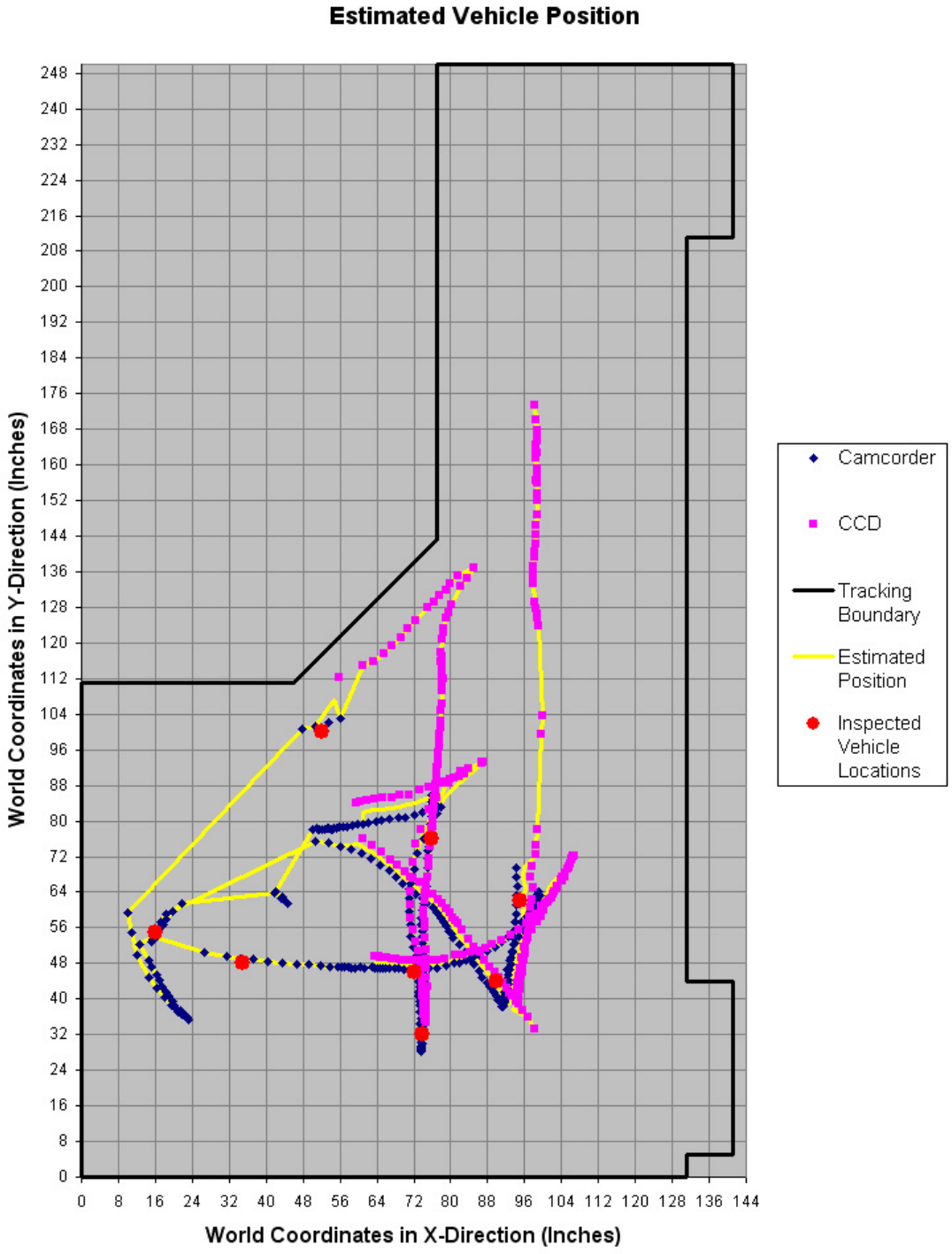

Figure 102: Calculated Vehicle Position and Estimated Path of RC Truck.

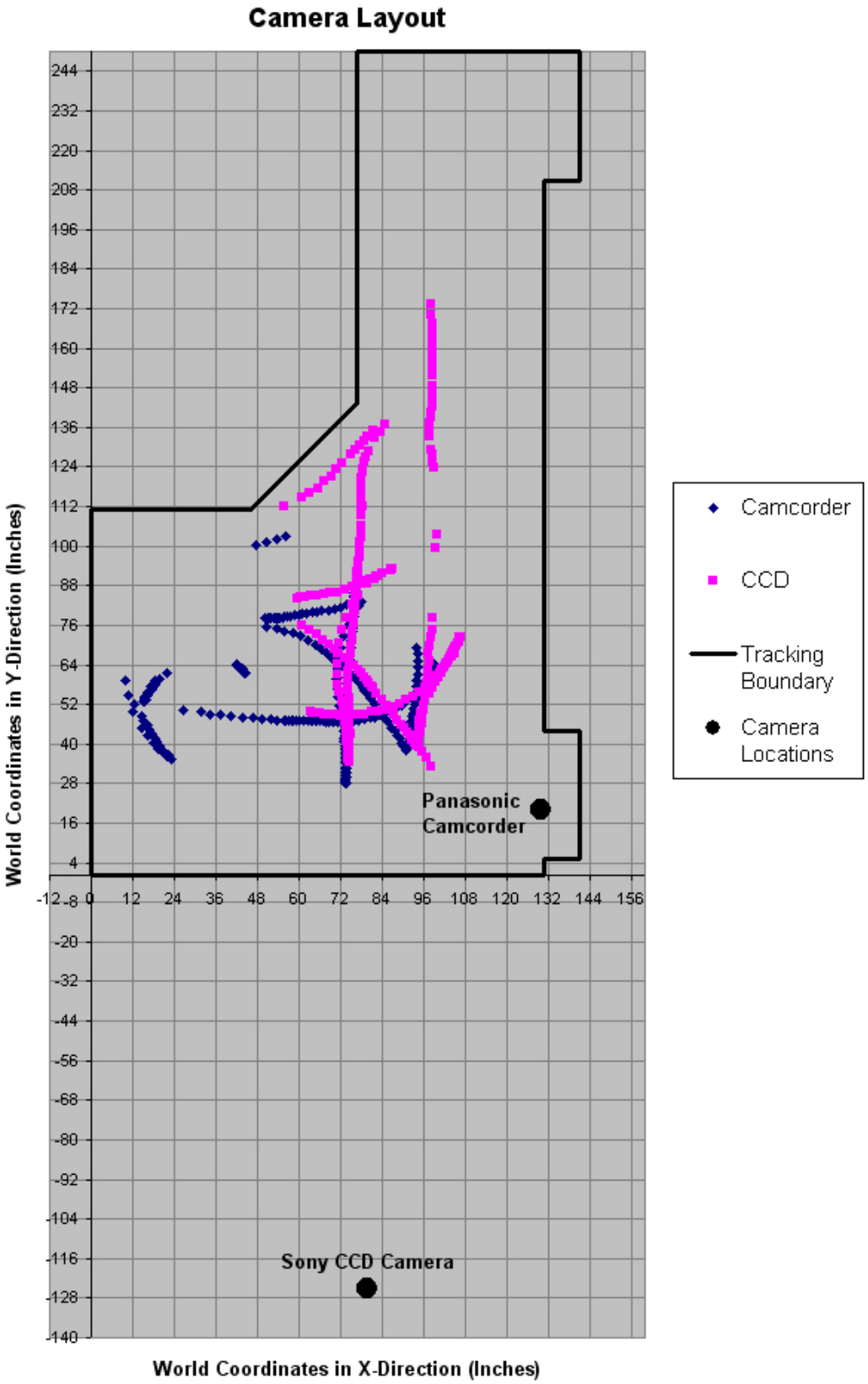

Figure 103: Location of Cameras with Respect to Tracking Area

Table 18: RC Truck Tracking Error

Frame No.	Examined X,Y	Measured X,Y	Error (%Error)	Error Synopsis
132	95.0 62.0	94.408 65.281	3.334 (3.9%)	---
209	72.0 46.0	73.524 48.374	2.821 (3.2%)	---
237	35.0 48.0	34.397 49.058	1.218 (2.0%)	---
380	52.0 100.0	50.547 101.351	1.984 (1.8%)	---
668	74.0 32.0	73.538 29.152	2.885 (3.6%)	---
698	76.0 76.0	74.045 75.931	1.956 (1.8%)	---
908	16.0 55.0	16.761 56.241	1.456 (2.5%)	---
964	90.0 44.0	88.542 42.630	2.001 (2.0%)	---

Test results

Eight frames from this test video were selected for tracking comparison between the calculated position and the inspected position. The errors in this case remain under 4%, which is not bad, but these values do not indicate some of the serious problems that occurred in this test. The video from this test was taken during a short window of opportunity, which did not allow for adequate testing of the vehicle tracking features. The poor tracking features, when combined the other common problems, yielded a fairly unsuccessful test. The problems encountered are listed in the following sections.

Inadequate feature classification

In the other tests, a few different color feature sets were tried for each test case. The colors that were most distinguishable from the rest of the image were used as the tracking features for that case. Because of the time constraints in creating this test case, no testing was done on the vehicle tracking features in the environment prior to taping. The features used were assumed to be adequate for this case. It was discovered, later on, that the tracking features work very poorly in this environment.

Figure 104: Poor Tracking Feature Color Selection

Figure 104 shows the vehicle on the tiled floor. The dark blue feature becomes very dark around its edges. When this color is used to model the first feature, the classifier tends to find all of the dark areas in the image as features. The second feature, the red circle, at a first glance appears to be unique in the image. On closer inspection, the flowers in the wallpaper are also a similar color of red. The second feature classifier tends to mistake these flowers as vehicle features. Finally, the background feature color, yellow, is very close to the color of the floor tile. Therefore, every part of the feature classification is non-unique in the image. To track this vehicle, classifiers had to be made very narrow, causing numerous misses when searching for tracking features

Problems with compressed video data

All of the same problems with image compression from the last test apply for this test case as well. The remote controlled truck used for this test is much quicker than the desktop rover vehicle, so interlacing was a much more significant concern in this case. Figure 105 shows a close-up view of the vehicle in a frame from this test. Even though the vehicle is only moving at a few miles per hour here, the interlacing artifacts are severe.

Figure 105: Interlacing Artifacts on Moving Vehicle

It would be advisable to re-run this test when more time is available. Better feature selection and lighting conditions would greatly improve the results of this test case.

Conclusions

The overall affectivity of this software exceeded expectations in many areas. When all aspects of the environment are precisely measured and good features are chosen, the accuracy of this position system can be extremely high. The high accuracy of this system when combined with ease of application to existing vehicles and inexpensive hardware, make the visual planar position system a powerful tool.

The accuracy of this system is very accurate when all aspects of the world model and camera information are determined correctly. The simulated test case shows that with this accurate information, the position information can be determined with less than

1% error. In the experimental cases, the camera information was not well known. X, Y, and Z coordinates of the camera could be easily obtained, but angular information was more difficult. Also, the camera fields-of-view were not easily obtained. Even with these uncertainties, minor adjustments could be made to the camera model to yield excellent experimental data. In the two experimental test cases, the error obtained at the measured positions of the vehicles was still relatively low. The error in the desktop rover test, which had the better camera models, stayed beneath 2%. The remote controlled truck case, where the camera model of the CCD camera was significantly flawed, still yielded errors less than 4%.

The speed of this system varies from case to case, but is typically quite fast. For instance, processing the simulation test case involves simultaneously processing two 640×480 video streams. Therefore, 614,400 pixels must be processed in each frame for this simulation. Moreover, the video files are uncompressed, requiring that the size of each frame on the hard drive be approximately 900kB. The testing of this software was done on a typical laptop, with very slow hard drives (4200 rpm). Even with the overwhelming number of slow downs and data bottle-necks, this software can still process those videos at around 10 to 15Hz. With the induction of live video the most substantial slow downs would be eliminated, thus likely allowing the processing of twice as many streams at twice the frame rate.

The main problem with this position system is the feature classification procedures. Some extra effort needs to be put into this area to ensure that vehicle tracking features can be found more regularly with less error. This final step would allow this system to accurately and reliably detect and track vehicles.

Another feature that would increase the functionality of this software is a camera calibration utility. The second most common errors came from incorrect camera models producing flawed position data. Software that could automatically determine the position, orientation, and field-of-view of the camera would make the system more robust and reduce setup time.

CHAPTER 7
FUTURE WORK

The monocular vision position system is complete and functional as per the writing of this thesis, but there are a few issues that could be expanded or improved upon. First, real-time video capabilities need to be added to the existing software. Second, the environmental conditions can be expanded to include non-planar surface tracking. Third, time dependent calculations should be added to determine velocity, as well as, position and orientation.

Real-Time Video

The most crucial addition to the existing software is including real-time video. This was originally planned to be part of this research, but was phased-out because of time constraints. Adding real-time capabilities is not as trivial as it may sound. This particular position system is reliant on multiple simultaneous video feeds. To achieve this in a working position system entails adding several video capture devices to the machine running the position system. Software must be written to enable real-time sharing of video memory for all cameras. Under the Linux platform, this will typically entail writing device drivers for each capture card, which is a daunting task.

A secondary option for processing real-time is to attach each camera to a separate computer. Each computer can then be interfaced through a local area network (LAN). This puts the full processing power of a stand alone PC on each video stream which would greatly improve speed. The difficulties come in sharing this information with other computers on the network. A host-client program must be written to specify which

system has the top-level control over the network and will calculate the final position and orientation information from the combined results from each PC. Again, this is an intimidating task that could easily become a master's thesis in itself, but would be the ideal solution to processing real-time video from multiple feeds.

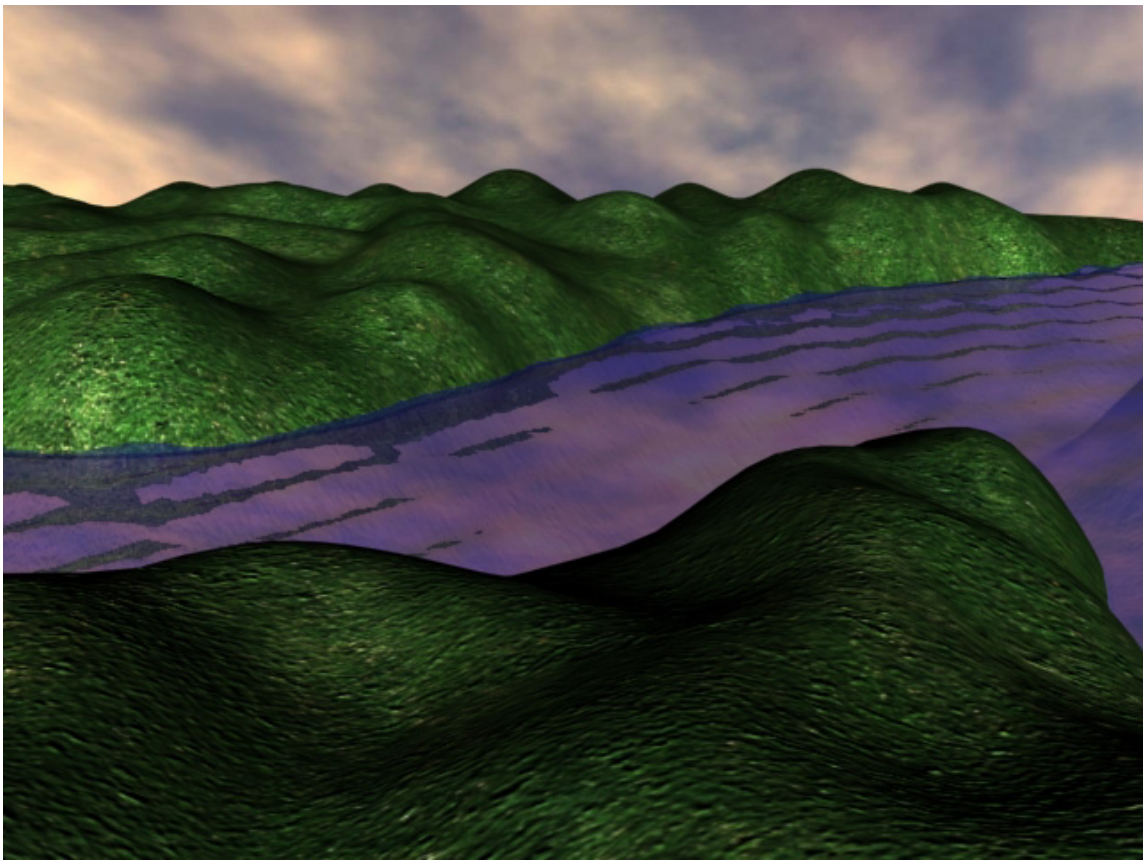

Figure 106: Example Non-Planar Vehicle Environment

Surface Positioning

The main problem with planar visual positioning is that vehicles have to be in an environment that has a planar driving surface. This constraint does not allow for much diversity. The only scenarios that will typically work with this system are located indoors. A method of expanding this system to non-planar surfaces was considered necessary so that outdoor environments could use this software, see Figure 106.

A method to do this was conceptualized while working with some computer aided drafting (CAD) software. Most CAD packages have 3D functionality that allows the user to design and view objects in three dimensions. Building 3D objects can be done in a number of ways. Most of the high-level design packages that are used in industry for design and fabrication use parametric modeling. Parametric modeling involves designing an object through a hierarchy of parameters that shape and mold the object with a set of strict guidelines. This is essential for designing a part that will eventually be fabricated.

Other packages, that are used for visualization more than fabrication, use a great deal of object primitives (spheres, cones, boxes, etc.) and systems of geometric Boolean operations (addition, subtraction, intersection, etc.) to create 3D objects. These CAD packages usually allow for much quicker generation of 3D models, but lack the precision and tools that are needed to be capable of fabricating the models.

These visualization based modeling packages typically handle 3D objects by viewing them in terms of their surfaces instead of their volumes. Viewing the objects in this way allows for simpler interactions between objects and faster display times. There are two main types of surfaces used in the visualization-type 3D modeling: polygon meshes and Non-Uniform Rational B-Spline (NURBS) surfaces. These two surface concepts are the root of non-planar positioning.

Polygon meshes are the more primitive of the two surface types, consisting of a number of data points connected by flat triangular or rectangular surfaces. This type of representation requires a lot of processing and data storage to account for all of the individual data points. Complex polygon mesh surfaces are typically more detailed than necessary, causing lengthy render times. This happens because multiple mesh polygons

have to be rendered to single pixels. The final color for one of these pixels is found by rendering all the mesh elements and combining their values.

This problem is addressed by using NURBS as an alternative type of surface model. NURBS surfaces are superior to polygon meshes in many ways. Each surface is defined by a two-dimensional array of continuous equations. A pixel intersection with a NURBS surface can usually be found as a single data point, minimizing the need for multiple calculations for a point. This makes rendering run much quicker and more efficiently. In addition to its efficient rendering properties, NURBS surfaces can be joined, broken, and manipulated easily and intuitively. Detail on how these things are done is outside of the scope of this research, but it is beneficial to know the basic properties of NURBS.

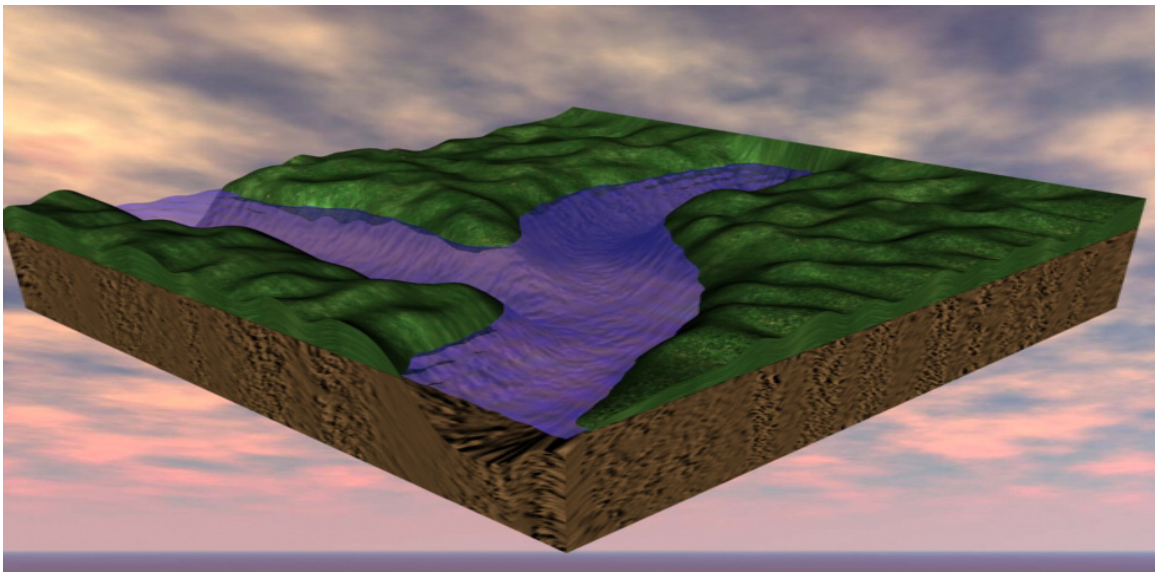

Figure 107: Example Non-Planar Vehicle Environment (Aerial View)

Both NURBS and polygon meshes are useful in the transition from planar to surface positioning. Building the 3D model of the driving surface is the only real difficult part of this transition; all the other calculations stay close to the same. The non-

planar environment shown in Figure 106 is displayed as a total tracking area in Figure 107. This is an exaggerated example of a possible vehicle environment.

A 3D model of the existing environment must be built so a driving surface can be defined. First, this area needs to be surveyed so that a detailed elevation map can be drawn. Surveying can be done by typical manual surveyor's tools or by more advanced methods like GPS. The detail will need to be fairly good to ensure good tracking results.

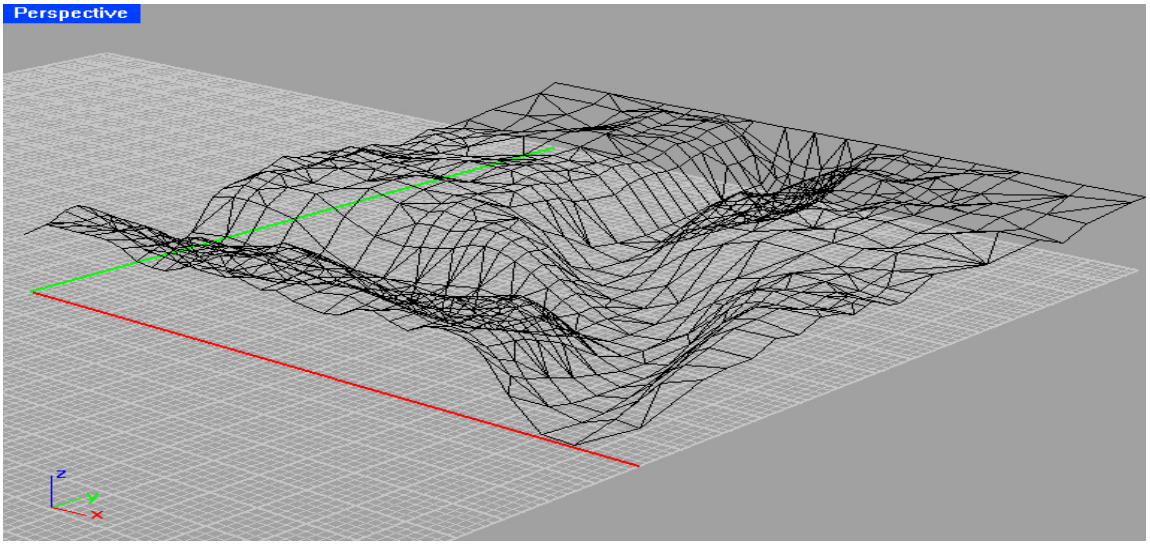

Figure 108: Polygon Mesh Built from Tracking Area

Once the area has been surveyed, the data can be used to create a polygon mesh of the area, see Figure 108. This mesh shows a number of mesh points where data would have been taken in the real-world environment. Since this mesh is simulated, the data points are in very good locations to show the land forms actual shape. This may not always be the case in a real environment, so it is good to try to get as many data points as possible.

Once this polygon mesh has been defined, it needs to be converted to a NURBS surface, see Figure 109. An algorithm that can take the polygon mesh point data and create the best possible NURBS surface will be essential to getting the surface

positioning to work. Surface data represented in NURBS will be easier to work with because of the inherent capabilities of working with continuous equations instead of individual data points.

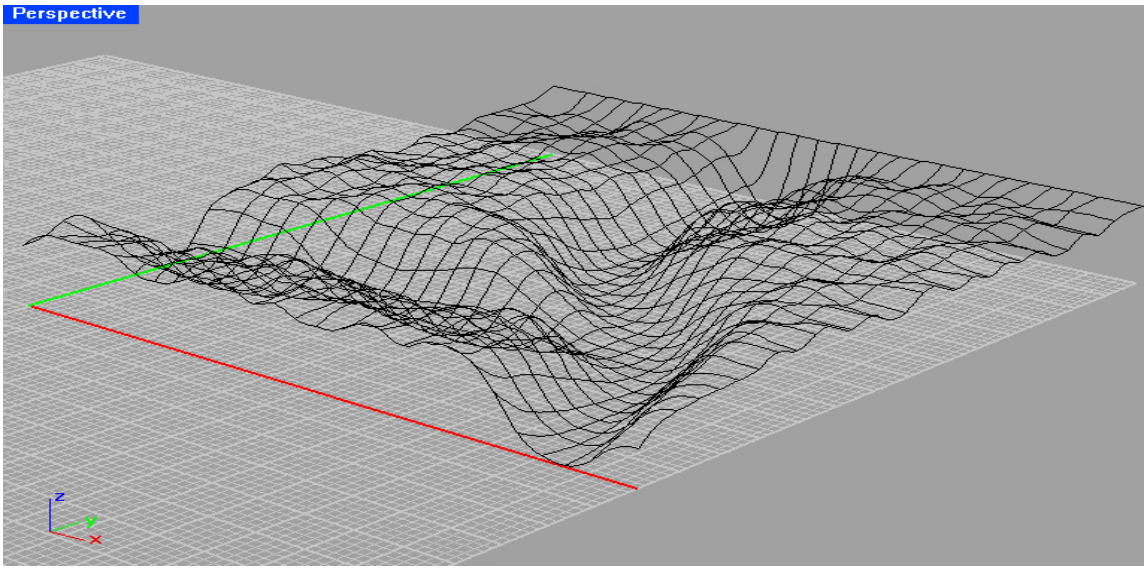

Figure 109: Polygon Mesh Converted to NURBS Surface

The final part of creating surface positioning data will likely be the most difficult. Once the NURBS surface is made, it needs to be offset in the vertical direction to create the surface that the vehicle features will be in. Offsetting a NURBS curve can create a problematic condition called a cusp. This occurs when a concave portion of the surface is offset at a distance larger than the radius of that portion of the surface. The result from the offset is a region of the new surface that overlaps itself, see Figure 110. This image shows a concave bowl shaped surface on the bottom with the offset surface shown on top. The bottom of the new surface has a flat round part surrounded by a sharp edge that is not present in the original surface. This happened because the bottom of the bowl offset correctly, but the part of the surface where the indentation starts to curve offsets too short. The ridges of the bowl on the sides, then over take the shorter offsets resulting in what appears to be an edge. The resulting surface may look discontinuous from the top,

but the surface equations actually create a loop underneath this edge surface that can be seen from the bottom. This portion of the surface is the cusp.

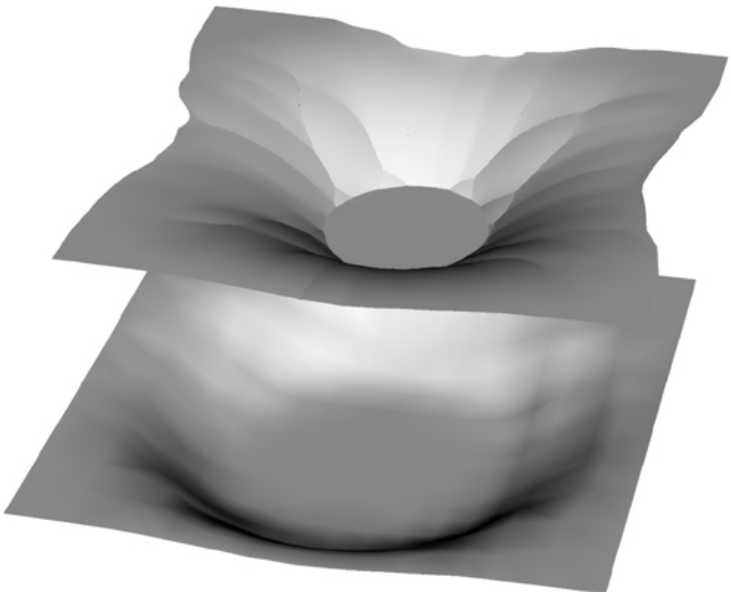

Figure 110: Cusp in an offset NURBS object.

A cusp in an image represents a part of the image in which the position data of the vehicle can be more than one point. This ambiguity of data at a surface cusp is another major concern of converting the planar position system to a surface based model. Of course, data cusps can be avoided if the ground does not make any steep curves and the tracking feature of the vehicle is fairly close to the ground. To avoid creating a surface cusp, it is important that the vehicle feature height not exceed the radius of any concave portions of the ground surface.

The final consideration that must be dealt with when migrating to surface positioning is the chance of having the principle axis of the camera pointing parallel to the vehicle feature plane. The problems that arise when this happens are similar to the problems caused by a surface cusp. A single portion of the image will represent more than one possible location of the vehicle, see Figure 111. The only time this occurs in planar positioning is when the camera is mounted at the same height as the vehicle

features, and is therefore can be easily avoided. In surface positioning, this occurrence is much more difficult to plan for and must be dealt with accordingly.

Figure 111: Vehicle Moving Parallel to Camera Axis

Once all of the problems have been dealt with, the resulting surface should accurately represent the non-planar surface that the vehicle features travel in for the entire tracking area. At this point the process is similar to the planar vehicle tracking. Instead of determining the pixel vector intersection with a single plane, the intersections must be found on the NURBS surface. The each of these intersections are mapped into a look up table for reference when processing the video.

Vehicle position is not the only information that can be obtained through the surface mapping model. Other constraints can be added to the map to keep the vehicle from traveling into areas that are off limits or impassable. Once such algorithm was developed when the concept of surface tracking was in its infancy. It was discovered that the vertical angle of each element of the polygon mesh could be easily found. This allows for the clipping of polygon mesh elements that are too steep for the vehicle to travel over. These elements can be removed from the mesh prior to building the NURBS surface so that the data at these locations is null, see Figure 112. This feature can be a powerful addition to surface positioning that helps keep the vehicle out of danger.

Surface is the next step in the evolution of monocular visual positioning. This addition could greatly increase the functionality of the system and is highly recommended for further investigation.

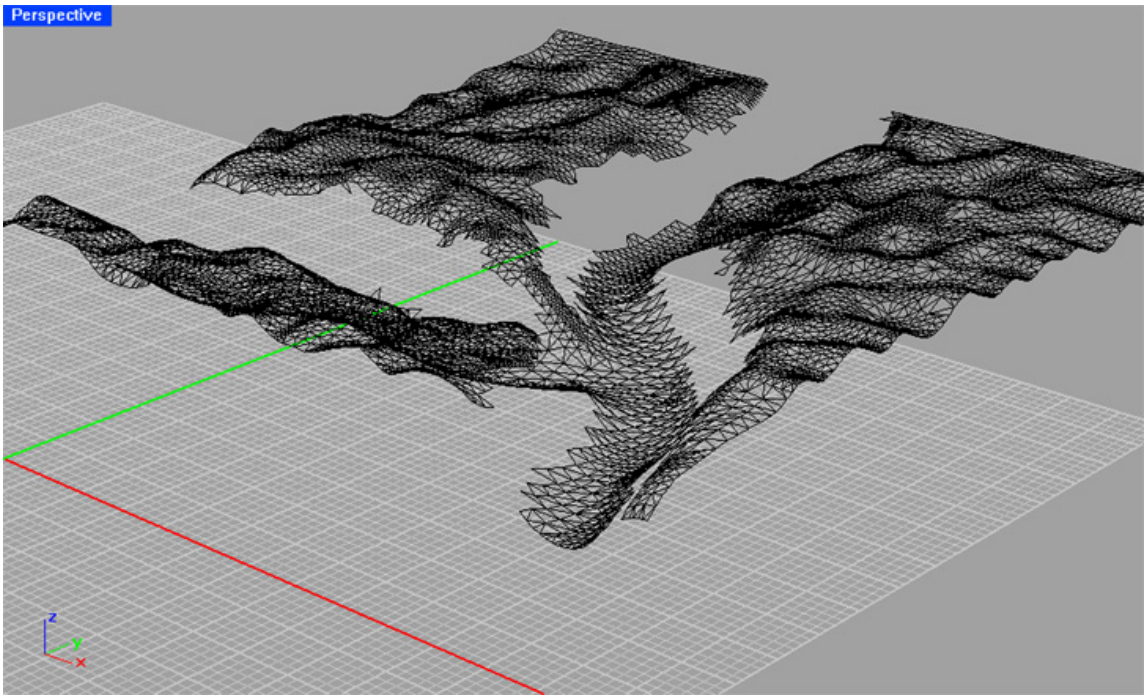

Figure 112: Polygon Mesh Surface with Steep Elements Removed

Velocity Tracking

A secondary application of visual positioning is the concept of velocity tracking. This is a simple and intuitive expansion of monocular visual positioning that could increase the functionality of the system. Once the world-coordinate positions of the image pixels are known, there is little work that needs to be done to find the velocity of an object moving in the image.

This concept mainly pertains to objects moving in a straight line through the viewing volume of the camera (i.e. boxes on a conveyor belt, vehicles on the highway, etc.) Figure 113 depicts boxes moving along a conveyor belt. The left image shows a snapshot of the conveyor belt with a box that has just come into frame. The second

shows the next consecutive image from the system where the box has moved about 1½ feet along the belt. If the video was taken at two frames a second, which is fast enough for this type of application, the box will be moving at 3 feet/sec. The location of the box can be measured by knowing the location of the camera and the height of the conveyor belt. The calculations are then exactly like the vehicle tracking, but the bottom edge of the box is being looked for instead of a specific tracking feature.

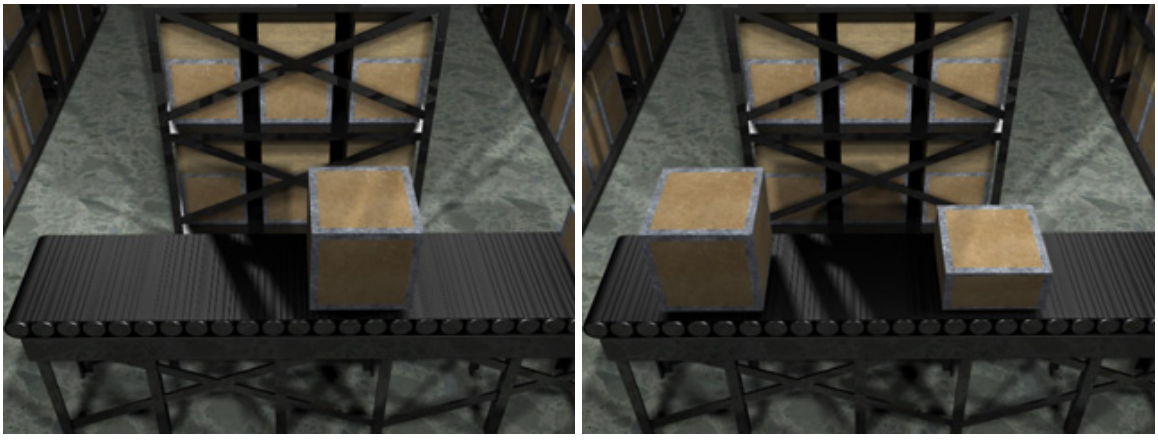

Figure 113: Boxes Moving on Conveyor Belt. First Frame (Left). Second Frame (Right).

Tracking speeds of highway vehicles works in much the same way, except the video must be processed much quicker. For vehicle tracking, the same camera can be used to determine the speed of the car and read the license plate of the car when it passes, see Figure 114 on page 211. In this figure, the vehicle can be seen entering the frame in the top image. The vehicle moves about twenty feet by the time the middle image is taken. If this video stream is taken at thirty frames per second and the second image is taken five frames after the first, the vehicle is moving at right over eighty miles an hour. The third and final image shows a close-up of the vehicle's tag, taken by the camera when the vehicle is determined to be speeding.

The advantage of this type of speed detection system is that there are no devices that a driver could use to identify it. Conventional radar and laser speed detection devices are easily foiled by radar detectors and police vehicles can typically be seen from a distance. If a camera is mounted covertly, such as up a telephone pole or on a light post, the chance of detection by a speeding driver is unlikely.

These cameras could be mounted in places where speeding is common or in places of high concern, like school zones. Data from these cameras could be routed directly to a police vehicle or to a processing station where tickets can be given to the driver by mail.

These are just a few of the possible additions to this software that could greatly improve its functionality. There are no doubt more situations and functions that this software could handle, but these few are enough to show the diversity of this program.

Figure 114: Speed Trap Camera Usage. First Frame (Top). Second Frame (Middle). Close-up of Tag (Bottom).

APPENDIX A
GRAPHICAL USER INTERFACE

The vision position system is basically a software library with the functions needed to process multiple video streams and determine a vehicle's position in the field of view. To use this system effectively requires an interface that allows the user to set parameters and view results real-time. At first, a graphical user interface (GUI) was created in Microsoft's Visual Basic®. This GUI was rather simple but allowed for real-time changes to be made to a stream being processed while displaying the results to the screen.

Soon after the creation of the first GUI, the software was ported to Linux. Visual Basic will not run on a platform other than Windows, so a new GUI was needed. Taking the shortcomings of the first GUI in to account, a new and more powerful GUI was created in Motif that would run under Linux. Several crucial improvements were included in the current GUI that greatly increased the functionality of the system. The new GUI's features include, but are not limited to:

- A stable platform – The current GUI does a number of error checking routines to reduce chances of user triggered crashes.
- Modular functions – The image processing is handled in a way that allows for the functions to be called in any order and any number of times. Also, the modular function design makes it relatively simple to add new function types to the program.

- Run-time modification abilities – Any video processing function can be inserted, removed, modified, moved, or temporarily disabled, while video is being processed. This makes testing a score of algorithms on a video stream quick and simple.
- Save/Restore program states – The current status of the program can be saved and restored at any time. This keeps the user from having to enter information into functions again and again when the program is shut down.
- Multiple stream capabilities – This GUI can process more than one video stream at a time. This is perhaps the most important feature of the new GUI. The visual position system relies on the use of several video streams in most cases, so the ability to process multiple video streams is very important. For this reason, the current GUI was designed to handle up to 10 simultaneous video streams at once. Each stream is associated with a different input file and the properties for each stream are set individually.

Basic GUI Operation

A significant amount of time and effort was put into creating the GUI software in hopes that it would be powerful but intuitive to operate. A few different design plans were tried before a final was decided upon. The final design looks and works much like a spreadsheet. The program form is split into a number of rows that contain individual functions. This set of rows is referred to as the stack. The base form with an empty stack is shown in Figure 115. This is how the program appears when it is first opened.

At this initial state, only one function appears in the stack: ‘Start Main Loop’. This function is essential to program operation, and is therefore always present in the stack. This function serves as the separator between the two classes of functions: the

preprocessing functions and the loop functions, and it will automatically reposition itself to stay between these groups.

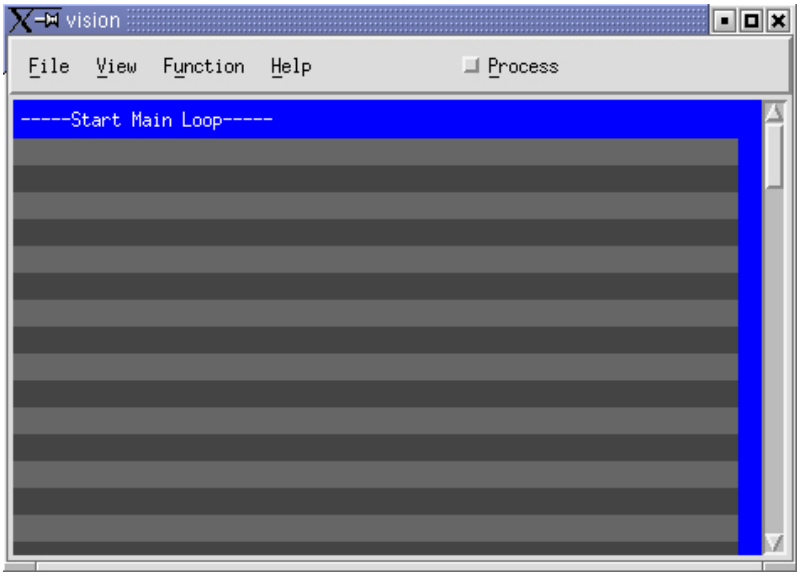

Figure 115: GUI Base Form

The alternating dark and light grey lines are placeholders in the stack for new functions. The number of placeholders is defined when the GUI is compiled, but by default there are 100 function placeholders in the stack. It is doubtful that more than 100 functions would ever be used in this software because of the excessive amount of time it would take to process this many functions.

Most of the GUI operations are handled by functions in the stack, but a few of the control options fall outside of the vision system parameters and have to included in the menu system. The menus and their associated functions are listed below:

- File menu
    - Save Properties File – This command saves the current program state to a binary data file. This data file contains all of the information from all of the functions currently loaded in the GUI.

  - Load Properties File – This command retrieves the saved GUI properties file. On load, all the functions are restored to the state they were in when the properties file was saved.
  - Exit – The command exits the program.
- View Menu – This menu used to have commands to control the video display windows. Now this is done via functions in the stack, therefore all the commands in this menu are obsolete. This menu will be removed in the next update of the GUI software.
- Function Menu
  - *Add Function* – This command opens the 'Add New Function' dialog box. From here any available functions can be added to the stack.
  - *Update All Functions* – This command updates all the functions in the stack and loads the function data into program memory. This is primarily needed when the user changes the value of a textbox in a function manually. The new text information is not finalized until the function is updated.
  - *Clear All Functions* – This command removes all the functions from the stack and from program memory.
- *Help* Menu – The help files and 'About' dialog have not been added as of this version of the GUI.
- *Process Button* – This button starts the vision system. When the vision system is running the button text will change to 'Stop' and the square icon will light up red. If pushed again, this button will halt the vision system.

Save/Load Properties Files

The ability to save and retrieve the program state is essential when using this software. Completely setting up the properties for a multiple camera tracking system can take a significant amount of time, so it is intrinsic that this information can be saved are restored at a later time. The program information is stored in the file includes some basic program properties along with all of the information in all the functions in the stack.

When either the <Open Properties File> or <Save Properties File> command buttons are hit, a file-search dialog box appears, see Figure 116. This dialog can be used to navigate the file system to find the desired input or output file. Pressing the <OK> button will use the text in the 'Selection' textbox as the filename. If the file does not exist, the path and filename can be typed directly into the selection box. Pressing <Cancel> will leave program in its current state.

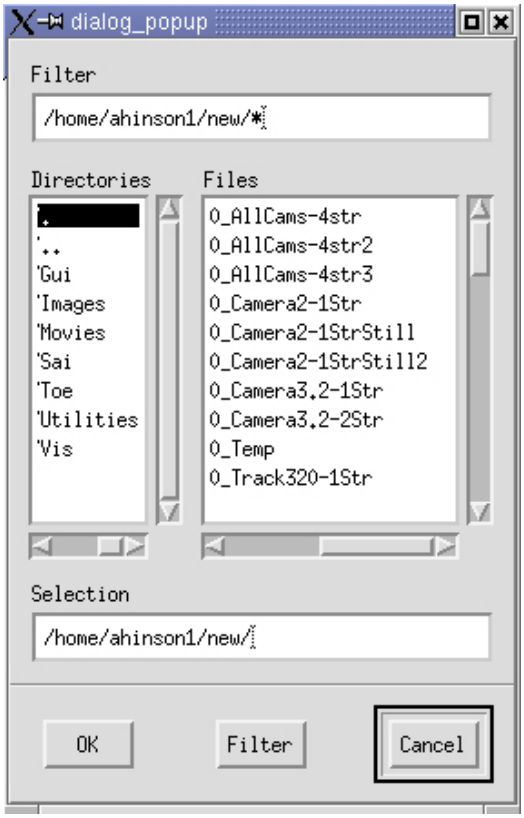

Figure 116: Save/Load Dialog Box.

Add New Functions

Selecting the <Add New Functions> command will pop up the 'Add Function' dialog box, see Figure 117. This dialog box has the list of available functions separated by genre. When the program is idle, any function can be added to the stack through this dialog, but when the position system is running, only loop functions can be added.

This selective availability of functions must be done to prevent preprocessing equations from being added after the preprocessing is complete. All file I/O functions

and definition functions are processed prior to any video data for efficiency reasons. It would not make sense to add one of these functions after processing begins, because the function would not ever get processed.

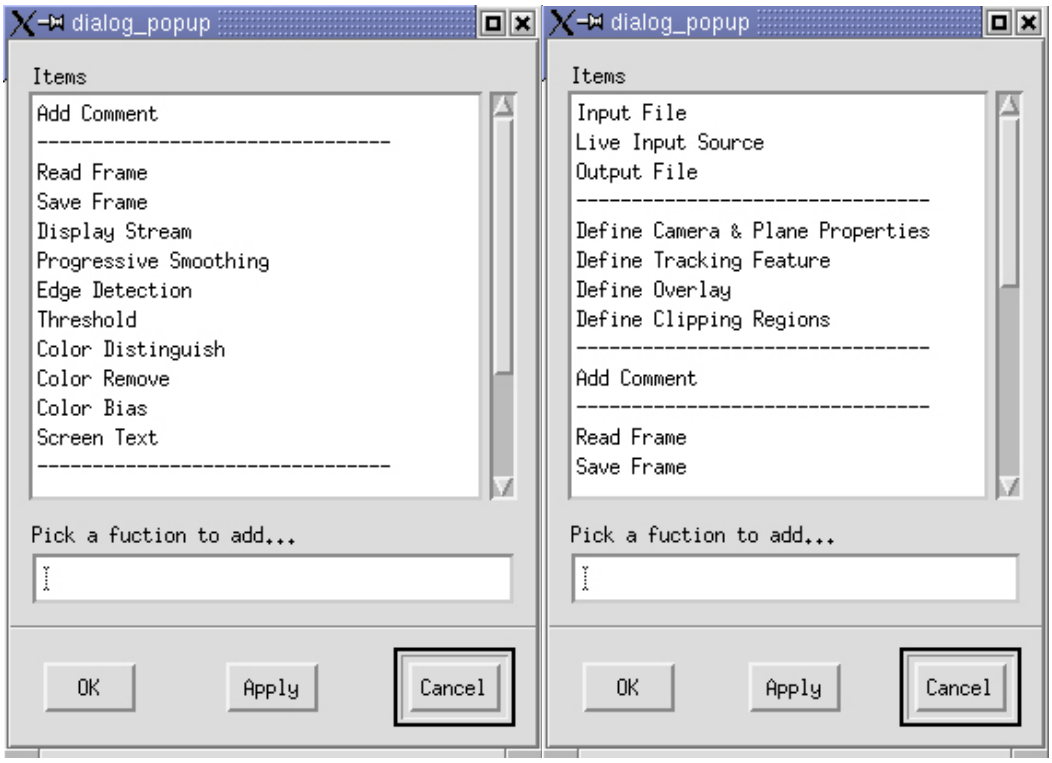

Figure 117: Add Function Dialog Box. Original (Left). Run-Time (Right).

Another reason for the selective availability is that all the error checking is done at the beginning of processing. If a user were to add another input stream while video was being processed, there would be no mechanisms in place to keep the user from crashing the system.

Process Button

The final menu item is the <Process> button. This command will start and stop the video processing. When the <Process> button is pressed, the vision position system initialization begins. Program initialization can take anywhere from a few seconds to a few minutes. This is the lengthiest portion of the program and cannot be stopped once it

is started. Once initialization is complete, the video will be processed frame-by-frame. At this point, the process button will read 'Stop' and can be pushed at any time to halt processing.

Function Layout

This program uses a standard form for all the functions in the program. Every location in the stack contains a function form with every field type on it. Each function uses different fields so the display must be different for each function. This is done by hiding the unused fields for a particular function, thus not allowing these fields to be set.

Figure 118: Example Functions Showing All Possible Field Types

Four example functions are shown in Figure 118. Between these four functions, all the possible fields are shown; field explanations are given in Table 19.

Table 19: GUI Button Functions

Field Name	Description
Function Number	This number represents the location of the current function in the stack. The function number is located at the far left side of every function. This number is mainly used in error checking. If a function is found to have an error, the error dialog is displayed with a description of the error and the number of the function that has the error. The only function that does not display a function number is 'Start Main Loop'. This function still occupies a slot in the stack and therefore has a number, but it is not displayed on the screen.
Function Disable Button	The function disable button is to the immediate right of the function number. This button allows the user to turn a function on and off in the stack. Only image processing functions have this capability. The functions that are run during preprocessing can not be turned off because they are essential to the operation of the system. This is mainly a convenience feature that enables the user to disable a function at runtime without having to physically remove it.
Stream Number	The stream number identifies to the system which video stream the current function is associated with. Every function must be associated with an existing video stream or an error will occur. Processing functions will only be run on the video file that is associated with the same stream.
Function Name	This is the text ID of the current function. Every function displays a function name.
Color Selector Button	The color selector button is present in functions that require values for red, green, and blue color components. This button displays the color indicated by the values in the red, green, and blue fields. The primary function of this button has not yet been implemented. Eventually, this button will be used to open a color palette dialog box to allow a color to be visually selected and have the red, green, and blue values updated automatically.
Integer Variables	Integer variables are common in the video processing functions. Integer value textboxes are only large enough to type three characters in. This is because the textboxes correspond with unsigned 8-bit integer values that can only run from 0 to 255. The most common use for the integer textboxes are red, green, and blue pixel values. Practically all of the primitive image processing functions require entries for these values. It is important to note that the values contained in these textboxes are only posted to the system when the <Update All Functions> or the <Update> button associated with that particular function is pressed.

Table 19. Continued

Field Name	Description
Floating Point Variables	Very few functions require floating point values. The main function that uses the floating point textboxes is the 'Define Camera & Plane Properties' function. Textboxes intended for floating point values are significantly larger than the integer textboxes, holding 12 characters instead of 3. When a floating point textbox is displayed without data, it will be filled with the null value '0.000000'. As with the integer textboxes, values entered into these textboxes must be manually updated.
String Variable	There is only one string data textbox in the function layout. This textbox is recognizable by its larger size (about twice the size of the floating point textboxes) and because it is always located by itself. This textbox typically holds control character data and sometimes text string data. Again, data in this textbox must be manually updated.
Filename String	The filename string textbox is identical to the string variable textbox, except that a <Browse> button is located at the top-right corner of the textbox. This textbox is used for entering filename and hardware address data. The text in this string must be updated by the user if entered manually. This process is automated if the <Browse> button is used.
Filename Browse Button	The filename browse button opens the filename dialog box (shown in Figure 116 on page 216). This dialog can be used to search for an existing file or to create a new file. Once the <OK> button is pushed, the path and filename displayed in dialog will be added to the filename string textbox.
Function Options	A few of the processing functions can be run in a variety of ways, so this field controls the settings of these functions. Only one of these options can be selected at once, so when an option is selected, the other options are automatically deselected.
Up/Down Buttons	These buttons allow the current function to be moved up and down in the stack. The preprocessing functions can not be moved, so only the functions located below 'Start Main Loop' can be rearranged. Since the functions are processed in the order that they are located in the stack, changing the order of the functions will affect the final result of processing. Any function with the up and down buttons can be moved throughout the stack from the position directly underneath 'Start Main Loop' to the position of the last entered function. Trying to move beyond this range will cause the program to display an error.

Table 19. Continued

Field Name	Description
Update Button	This button updates the function values on the screen to the program's memory. This button is present on all the user-added functions in the stack. When the processing is started, the update button for preprocessing functions disappears so that the user cannot change critical function values and cause errors. The update button stays present for the functions after 'Start Main Loop'
Remove Button	The button removes the given function from the stack. When pressed, the current function is removed and functions located lower in the stack move up to fill the gap. Like the <Update> button, the <Remove> button disappears for the preprocessing functions when the processing begins.
String Variable Explanation	On the functions that take control strings, the characters that are used are not always intuitive. The explanation of what control characters are used (and for what) is shown in this label.

Available Functions and Processing Order

Using the GUI may seem daunting at first, but the layout is quite simple. The most important thing, is to know what functions must be added to be able to process video. The needs will change depending on the type of processing being done, but transition from one style to the next should not be too difficult.

First, the layout of the functions in the stack needs to be examined. The function stack can be broken down into two parts: the preprocess functions and the loop functions. The preprocessing functions are located at the top of the stack and the loop functions are at the bottom. The two regions are separated by the 'Start Main Loop' function which defines the start of the loop equations.

Any time a new preprocess function is added, it is placed at the top position of the stack, pushing all the other functions down. The order of the preprocessing functions is irrelevant to the program so their position cannot be changed once they are inserted.

These functions can only be modified and removed when the system is idle. Once processing begins, the <Update> and <Remove> buttons for these functions disappear.

The loop functions consist mainly of the video processing functions. All of these functions must all be located below the 'Start Main Loop' function. They are processed in the order that they are located in the stack, so position for these functions is important. These functions can be added, removed, and repositioned at any time.

Preprocessing Equations

The preprocess functions include any functions that must be run prior to handling video. These include stream I/O functions and definition functions. It is obvious that the I/O functions must be dealt with first, because without them there will be no stream to process. The definition functions are used to create the lookup table (LUT) and handle other calculations that will not change throughout the length of processing. Many preprocessing functions are dependent on other functions, so they must be processed in the order of their importance not in their physical order.

Input files

The first functions to be checked are the 'Input File' functions. These are the most critical functions in the program because they define the stream that will be used for all subsequent functions. Each 'Input File' function is first checked for errors, and then the associated input file is opened and initialized.

'Input File' is the only preprocessing function that is absolutely required for the system to run. The simplest processing example would be having 'Input File' as the only preprocessing function and 'Read Frame' as the only loop function. When live video capabilities are added to this software, the 'Stream Input' function can be used

interchangeably with ‘Input File’. The ‘Input File’ function can be seen in Figure 119 and the ‘Stream Input’ function is shown in Figure 120.

The fields for the ‘Input File’ function are self explanatory. The ‘Filename’ textbox contains the path and name of the input file and the ‘Str’ textbox contains the stream number. The layout of ‘Stream Input’ works close to the same way, except the ‘Filename’ textbox is referred to as the ‘Address’ textbox. For the ‘Stream Input’ function, the hardware address of the video capture card must be put into this ‘Address’ field.

The approach for connecting to a video card may change when this function is actually implemented in the code. If connecting to a video card becomes too complicated, the ‘Address’ field might be replaced with a ‘Configuration File’ field. This configuration file would have all the information necessary for the software to connect to the video capture device.

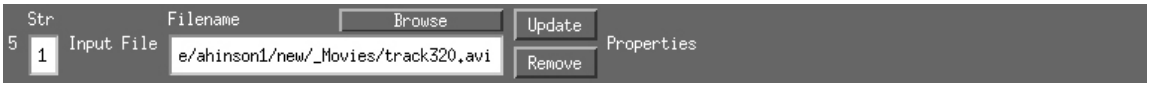

Figure 119: ‘Input File’ Function

Figure 120: ‘Stream Input’

Output files

The next to be processed are the ‘Output File’ functions. Output files are used to save the current frame data at any point in the processing. Output files must be associated with an existing stream to avoid causing an error, and only one output file can be define per stream. Multiple save points can be defined in the loop for each output file so that more than one video can be created from a single output file. The current frame, function number, and stream number are all saved for each image sent to the output file

so that the images in the file can be properly parsed and combined to create bitmaps or AVIs. Creating bitmaps and AVIs from the output files is accomplished using a separate utility. The 'Output File' function can be seen in Figure 121.

The fields for the 'Output File' function are identical to those in the 'Input File' function. The 'Filename' field for this function must specify a file that either does not exist or a file that does exist and is write-enabled. If the output file cannot be opened at runtime, an error is displayed and processing stops.

Figure 121: 'Output File' Function

Define camera & plane properties

Once all the stream I/O functions are processed, the software can start on the define functions. The first define function to be processed is 'Define Camera & Plane Properties'. This function gives the software intrinsic camera properties, extrinsic camera properties, and the ground plane equation. This data, when combined with the resolution of the video stream (which is obtained in file initialization), can be used to build the LUT. This part of the preprocessing can take a significant amount of time if the video resolution is high. After the LUT is built, it is checked for errors. Any values that are determined to be faulty are removed from the LUT. The 'Define Camera and Plane Properties' function is shown in Figure 122.

All the quantities for this function are floating point values. The first column of values relates to the camera's X, Y, and Z coordinates in the world coordinate system (WCS). The second column is the pan, tilt, and slant of the camera about its axes in the WCS. The third column is the intrinsic properties of the camera: field-of-view in the

horizontal and vertical. The final column refers to the homogeneous coordinates of the tracking feature plane. All values must be defined in a right handed coordinate system.

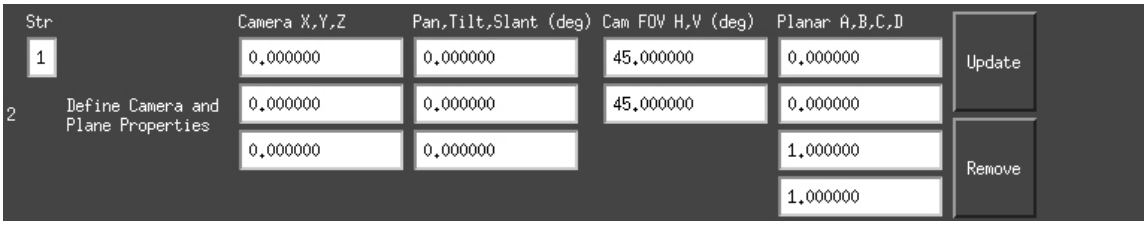

Figure 122: 'Define Camera and Plane Properties' Function

Define clipping regions

The 'Define Clipping Regions' functions must be processed next. The polygons specified in these equations will be used to remove data from the LUT that the user does not want included in the tracking area. Each function can describe several polygons that will be incrementally combined in Boolean fashion. The final result of the combination is an overlay that is used to remove unwanted regions from the LUT. The 'Define Clipping Regions' function is shown in Figure 123.

The 'Define Clipping Regions' function uses a control string to create the polygon clipping information. The definition of the control variables is shown on the far right. The control characters: 'A', 'C', and 'U' are stand alone commands. The other two control characters: 'I' and 'E' require polygon data, enclosed in parentheses, to be entered afterward. As with all textbox fields, the data must be updated before it is posted to program memory. More detailed information about how to define clipping regions is located in Chapter 5.

Figure 123: 'Define Clipping Regions' Function

Define overlay

Once the LUT has been built and the user-defined clipping regions have been removed, the LUT is in its finalized state. Since there are no more modifications that can be made to the LUT, overlays based on the LUT data can finally be generated. Any number of overlays can be added to a single stream, and they are assembled in the order that they are listed in the stack. The 'Define Overlay' function is shown in Figure 124.

The color of the overlay is defined by the 'Red', 'Grn', and 'Blu' fields. The 'Grid' field determines how many gridlines to show for range data overlays. The 'Overlay Set' field contains the controls strings for the overlay function. Multiple overlays can be built with a single function but they will all be the same color. Typically, it is recommended to have different overlays be different colors to avoid confusion. This requires that a separate 'Define Overlay' function be used for each overlay. The control strings for this function are shown at the far right.

Figure 124: 'Define Overlay' Function

Define tracking feature

The 'Define Tracking Feature' function defines the colors and distribution model of the colors being searched for in the image. This function is run in the preprocessor because it only needs to be done once. Since it has no bearing on any other preprocessed functions, it is the last function run in the preprocessor. The 'Define Tracking Feature' function reads **.trk* files that are typically generated by the Training Data Generation Program (This program is discussed in detail in Appendix C). The tracking data file tells the program which type of distribution to use for tracking the features (Color Range, 3D

Gaussian, etc.) and the optimal values to use with these distributions. The 'Define Tracking Feature' function is shown in Figure 125.

This function works exactly like the 'Input File' function. The filename of the tracking data file must be entered into the 'Filename' field and associated with an existing stream. Eventually, the capabilities for tracking multiple vehicles should be added, in which case, there could be multiple 'Define Tracking Feature' functions per stream. For now, only one of these functions can be defined per stream.

Figure 125: 'Define Tracking Feature' Function

Loop Equations

After running the preprocessor, the program is initialized and ready to manage individual frames. The loop equations are responsible for handling all of the work done to the video stream on a frame-by-frame basis. This includes modifying pixel data, reading and saving frames, and displaying frames to the screen. Functions in this part of the program will be run once per frame in the order that they lie in the stack. When the bottom of the stack is reached, the frame counter is incremented and the processing for the next frame begins. This process continues until the end of every stream is reached or until manually terminated.

As mentioned earlier, the order of the functions is crucial to determining the final effect on the video frame. At this point, all the data buffers are initialized and ready for a video frame. The loop functions are less likely than the preprocessor functions to cause problems so less error checking is done in this part of the program. This speeds up

processing time, but allows the user to make mistakes that will cause undesirable results to the video frame.

For instance, an image processing function can be run on a frame that has not yet been read. The frame buffer already exists, but before a frame is placed into the buffer, there is just garbage in the memory location. Therefore, an image processing equation will have bizarre results when running on this data. Also, the frame buffer is not cleared after each frame, so the last frame that was read will remain in the buffer until a new frame is read. If a processing equation is higher in the stack than the 'Read Frame' function, this equation will be constantly processing the previous frame instead of the current. These types of problems will not crash the system, but need to be watched for anyway.

Read frame

The 'Read Frame' function is a marker to tell the system to read a frame for the associated stream. This function's only parameter is the stream number. The frame to be read is determined by the number of main program loops that have been completed, not by the number of 'Read Frame' functions called. Therefore, every time 'Read Frame' is called in the stack, the same frame is read and put into the frame buffer.

Every stream must contain at least one 'Read Frame' function to enable processing. Any streams defined without a 'Read Frame' statement will cause an error and processing to stop. The 'Read Frame' function is shown in Figure 126.

Figure 126: 'Read Frame' Function

Save frame

The compliment to the 'Read Frame' function is 'Save Frame'. Every time this function is called, the frame in its current state is saved to the associated output file. Obviously, an 'Output File' function must be defined for this stream or the program will display an error.

'Save Frame' can be called multiple times in the stack for the same stream, which will save the current frame in multiple states to the output file. This 'Save Frame' function's information is saved to the frame, as well, so the individual frames can be split up by calling function. In this way, multiple video files can be made from a single stream showing the various stages of processing. The 'Save Frame' function, like the 'Read Frame' only takes one parameter: the stream number. The 'Save Frame' function is shown in Figure 127.

Figure 127: 'Save Frame' Function

Display stream

The 'Display Stream' function is responsible for displaying the video stream to the screen. This function is purely aesthetic in nature and does not modify the stream in any way. 'Display Steam' creates a popup window the same resolution as the video stream to display its contents. The display window is created the first time the function is called, and the display image is updated every time the function is called after that. There is a lingering problem with the display windows that do not allow the user to close them once they have been created. If the program tries to access a display window that has been closed by the user, a fatal *Broken Pipe* error occurs and the program shuts down. This problem is being worked on and will hopefully be resolved in the future.

The ‘Display Stream’ function uses two parameters: the stream number and the ‘Apply Overlay’ toggle. Each ‘Display Stream’ function is associated with one display window. Each display window shows the window ID, stream number, and the function number of the ‘Display Stream’ function that created it. The ‘Apply Overlay’ allows the user to enable and disable the overlays for the stream. This switches all existing overlays for the stream on and off. Overlays cannot be switched independently. The ‘Display Stream’ function is shown in Figure 128.

Figure 128: ‘Display Stream’ Function

Statistical feature tracking

The ‘Statistical Feature Tracking’ function tells the program at which point in the stack to search the input stream for the color data described in the ‘Define Tracking Feature’ function. The complete color model and algorithm are determined during preprocessing, but a call to the ‘Statistical Feature Tracking’ function is required in the main program loop to run a statistical search in the current frame.

When the ‘Statistical Feature Tracking’ function is run on the frame the entire image is searched for colors matching the specified model. Every cluster of correctly colored pixels are selected into a blob. When all appropriately colored pixels have been grouped into blobs, the worth of each blob is determined by checking a few different features. First, the border of the blob is examined. If the border contains mostly pixels that are the feature background color, then the chances of the blob being a vehicle feature are good. Second, the shape of the blob is examined. Since the tracking features are round, a blob must appear to be either circular or ovular to be a valid tracking feature. Finally, the overall size of the blob is checked. If a blob contains only a few pixels, the

chance of the blob being a feature is minimal. Using all of these factors as decision data, the best blob's information is returned so that its location in world coordinates can be found in the LUT.

Figure 129: 'Statistical Feature Tracking' Function

Color bias

The 'Color Bias' function modifies the color channel intensities of the red, green, and blue channels. Lowering the value of the 'Red', 'Grn', or 'Blu' fields of this function will diminish the associated color's presence in the image. This function can be used to strip colors from an image or modify the hue. The 'Color Bias' function is shown in Figure 130.

Figure 130: 'Color Bias' Function

Color distinguish

'Color Distinguish' searches an image for a color range and removes all pixels that do not display that color in the image. This function uses the fields 'Red', 'Grn', and 'Blu' to determine the target color. The 'Thr' field gives the threshold that should be used when searching for the target color. 'Color Distinguish' is shown in Figure 131.

Figure 131: 'Color Distinguish' Function

Color remove

'Color Remove' is the compliment to 'Color Distinguish'. 'Color Remove' searches the frame for the specified color and removes it from the image. The fields are the same as for 'Color Distinguish'. 'Red', 'Grn', and 'Blu' specify the target color and

'Thr' specifies the threshold range to search in. The 'Color Remove' function is shown in Figure 132.

Figure 132: 'Color Remove' Function

Edge detect

The 'Edge Detect' function searches the frame for image edges. These are found by taking the difference of consecutive pixels throughout the image and displaying the results in the frame. There are three different types of edge detection that can be done via this function: horizontal, diagonal, and dual. The differences in these three varieties are discussed in Chapter 2. The only parameters for the 'Edge Detect' algorithm are the detection type and the stream number. Only one type can be specified at any given time. The 'Edge Detect' function is shown in Figure 133.

Figure 133: 'Edge Detect' Function

Threshold

The 'Threshold' algorithm drops pixels whose values are below a cutoff range, and ignores the rest. The thresholding used in this software comes in a few different varieties. These different types of thresholding are discussed in Chapter 2. Like the 'Edge Detect' function, the type of thresholding to use is specified by the options field. Either 'Maximum', 'Stretch', or 'Drop' must be selected to indicate which threshold algorithm will be used. The color channel cutoff values for thresholding are specified by 'Red'. 'Grn', and 'Blu'. The 'Threshold' function is shown in Figure 134.

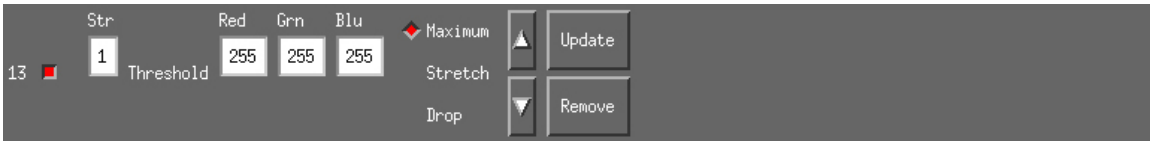

Figure 134: 'Threshold' Function

Progressive smooth

'Progressive Smooth' slightly blurs the video frame to minimize the color aliasing that occurs from image capture. The smoothing is done by bleeding color information from one pixel into its neighbors. The smoothing is done progressively in block of $n \times n$ pixels at a time. The larger the block of pixels the more the image will blur. The 'Amt' field of this function specifies how large the block should be. A blur amount of more than two or three will take a significant amount of time to process and is not recommended. The 'Progressive Smooth' function is shown in Figure 135.

Figure 135: 'Progressive Smooth' Function

Screen text

The 'Screen Text' function writes a text string to the screen. The color of the text is specified by the 'Red', 'Grn', and 'Blu' fields. The insertion point of the text in the image is obtained by the 'PixX' and 'PixY' fields. These values give the pixel location that the bottom left corner of the text string should be in. The justification option of the function has not been added at this time, but will hopefully be added in the future. The final field is the 'Message' string. The text message is typed into this field using the same control characters used in C/C++ for new lines, carriage returns, etc. The 'Screen Text' function is shown in Figure 136.

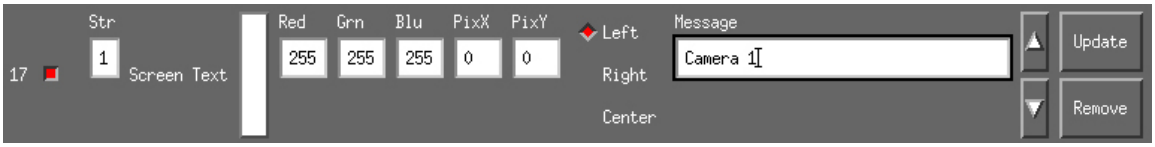

Figure 136: 'Screen Text' Function

Processing Examples

A few examples of image processing will now be done to show how to use the functions and the GUI to process video data. Both of the following examples are taken from one of the simulated warehouse videos created to test this software. This warehouse has three different cameras located around the tracking area. The first example will show how to process a single data stream using one of these camera views. The second example will show how to process multiple data streams by using the other two camera views. The function setup may seem a little overwhelming, but the logic is fairly simple.

Single Stream Processing

The single stream case is the less complicated of the two, so it will be discussed first. Program stack for this example can be seen in Figure 137 on page 239 and the video output to the screen is shown in Figure 138 on page 240. This example has all the functions needed to set up a well defined tracking area. Every function in the stack is connected to a single video input, stream #1. The functions in the stack are as follows:

- Input File (*Fcn 5*) – This function defines the input file. In this case, the video stream is simulation video from the camera in the rear-right of the warehouse model. This video stream has a resolution of 640×480 and 501 frames.
- Define Camera and Plane Properties (*Fcn 4*) – First of all, this function gives the extrinsic and intrinsic camera information to the system. The world coordinate location of the camera is listed (in feet) in the first column. This software does not use any particular units, but it is required that all comparable measurements are in the

same units. The second column is the orientation of the camera. This particular camera is panned 26.5° clockwise from the home position, tilted up 73.5° from the downward position, and is not slanted about the focal axis. The focal length of this camera is 35mm with an aspect ratio of 4:3, so the fields of view in the horizontal and vertical are 54.432 and 42.185, respectively. The final information obtained by this function is the equation of the plane. The ground plane is normal to the vertical axis of the world coordinate system and the vehicle tracking features are 5 feet above the ground. The equation for the tracking plane equation is, therefore, [5;0,0,1].

- Define Clipping Regions (*Fcn 2*) – The warehouse simulation has a well defined area that the vehicle is allowed to travel in. For this reason, the out-of-bounds areas need to be clipped from the lookup table to ensure the vehicle does not venture into regions containing obstacles. This function takes the parameter string:

  ci(-9.996,-7.463 ; -9.996,34.104 ; 40.039,34.104 ; 40.039,-7.463)

  e(-4.623,-3.589 ; -4.623,4.412 ; 35.379,4.412 ; 35.379,-3.589)

  e(-4.623,16.411 ; -4.623,24.412 ; 35.379,24.412 ; 35.379,16.411)

- This string tells the program to first clip all available areas from the LUT with the ‘c’ token. Then the ‘i’ tells the system to expect polygon information and to add this area to the available drive path. The polygon data is contained in parenthesis and comprised of x-y coordinate pairs separated by semicolons. Each coordinate pair represents a vertex of the polygon. For this polygon, the data represents the four corners of the warehouse, and therefore, includes the entire indoor floor space area. The next two polygons are preceded by the ‘e’ token, which tells the system to remove the polygons from the available map. Each of these polygons corresponds to one of the storage racks in the warehouse. Once these operations are finished, the

final map is complete. The LUT now only has data corresponding to areas inside the warehouse that the vehicle can travel in.

- Define Overlays (*Fcns 0 and 1*) – These two functions define the tracking area overlays in the image. These two particular overlays display a series of x and y grid lines on the display. The location of these lines is determined from the data in the LUT. Clipping out data in regions prior to applying the overlays causes the gridlines to only appear in regions that are included in the tracking area. This effect can be seen in Figure 138 where the storage racks cause breaks in the gridlines. Keep in mind that the grid is located several feet off of the floor so the gaps in the grids are located at this height as well.
- Define Tracking Feature (*Fcn 3*) – The color distribution of the vehicle tracking features is described in this file this function references, 'output.trk'. The vehicle features in this simulation are cyan and green, both on a dark blue background. For this stream the algorithm used is *color range*. *Color range* typically works the best for finding features in a simulated video file.

This concludes the preprocessing functions. At this point the files have been initialized, the LUT has been built and modified to only show relevant locations, and the overlays have been created. Now the main program loop is started. The loop will run for 501 iterations (the number of frames in the input file) or until manually stopped. Each of the following equations is processed for every loop in the given order:

- Read Frame (*Fcn 7*) – This is the first function executed in every loop. It is important to have the 'Read Frame' occur before processing or displaying a stream, to ensure

that there is pixel information in the frame buffer when the processing equations are run.

- Statistical Feature Tracking (*Fcn 8*) – This function handles all of the image processing that occurs in the stream. This function tells the program to search for the tracking features described in 'Define Tracking Feature'. When an appropriate feature is found, the location of the feature in the image is highlighted yellow. The border pixels are red and green, indicating the pixel is a non-feature color or a background color, respectively. If the tracking features are defined effectively, there should be more green border pixels than red on a tracking feature.
- Display Stream (*Fcn 9*) – The final function in the loop is to display the processing results to the screen. The final image should show the original video footage with the vehicle features highlighted in yellow and the x and y gridlines overlaid in magenta and cyan. The display results are shown in Figure 138.

Multi Stream Processing

The multiple stream example is practically identical to the single stream example, except that there are twice as many functions. In this example, the streams are being processed with the same equations that were used on the single stream. The only differences in these streams are the cameras. Obviously, each camera has a different location and orientation, but the camera type changes as well. The single stream example uses a camera with a 35mm focal length, but the two cameras in the multiple stream example use 24mm focal lengths. The shorter focal length gives the cameras a wider field of view.

The overlay definitions for x and y gridlines were left the same for all three streams so comparisons can be made between each display. The x and y coordinates of objects stay consistent from camera to camera, although it may be difficult to directly see this because of the different perspectives that each camera has. The function stack of the multiple stream example is shown in Figure 139 on page 241, and the output displays are shown in Figure 140 on page 242.

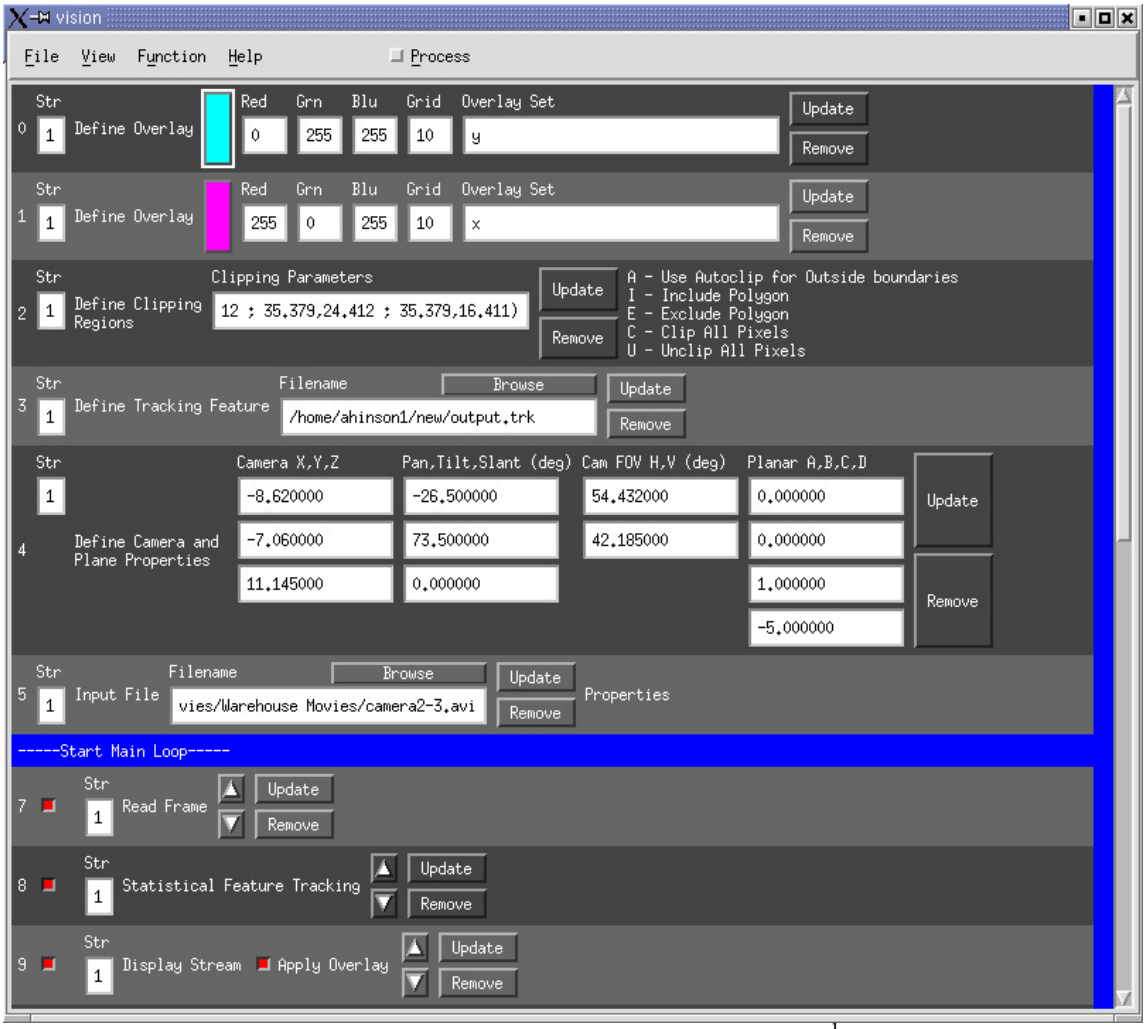

Figure 137: Single Stream Processing Example, Program Stack.[1]

[1] The 'Define Camera and Plane Properties' shows the 'D' value of the planar equation to be equal to -5 instead of 5. This is due to a glitch in the program that requires the offset distances to be negative. This problem will be fixed in a later version of the software.

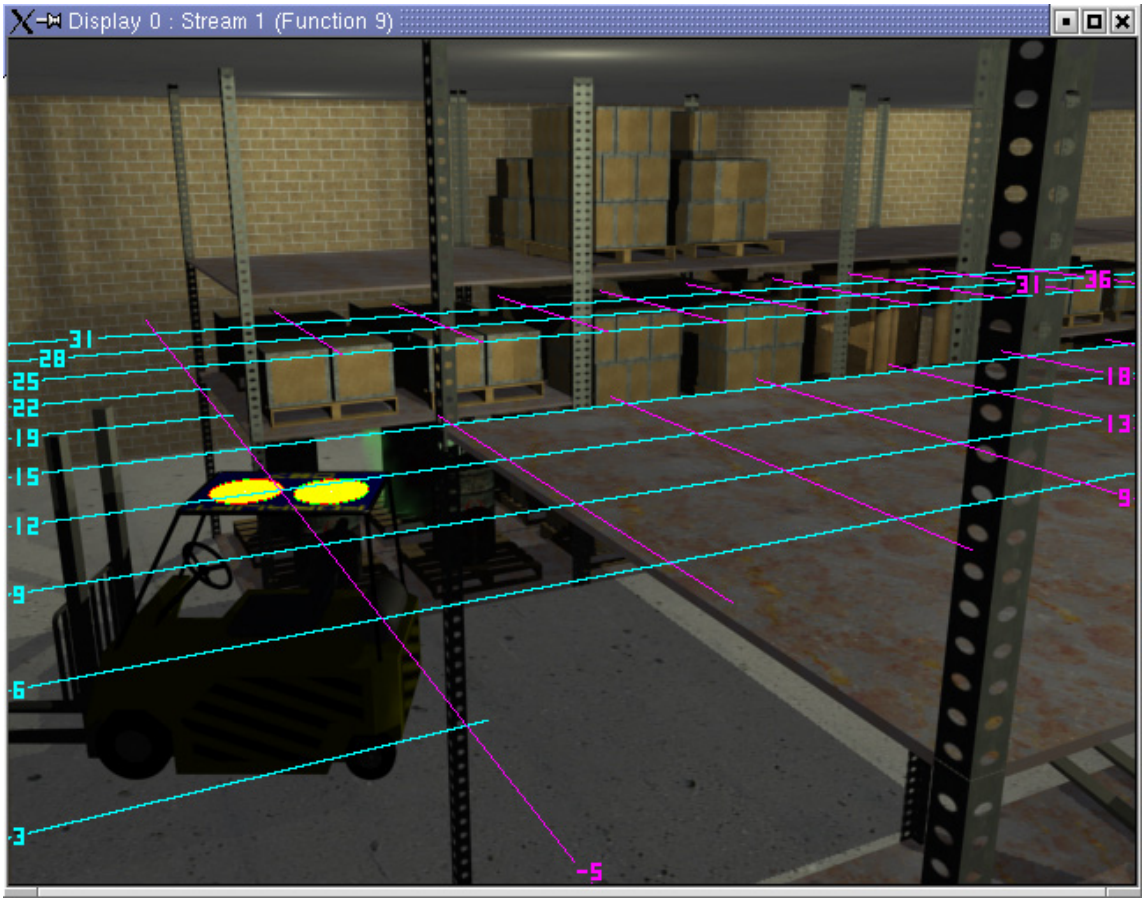

Figure 138: Single Stream Processing Example, Video Display.

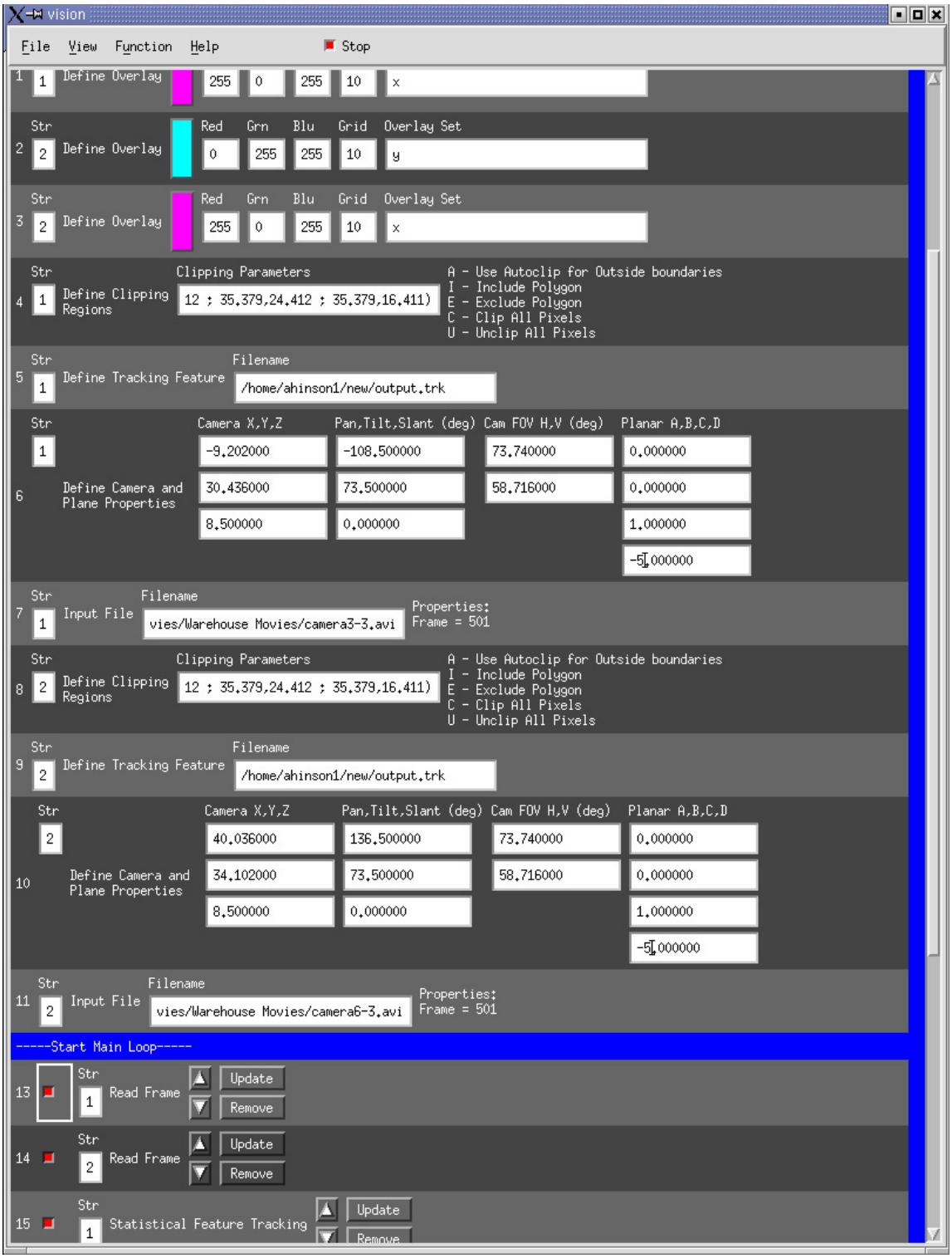

Figure 139: Multiple Stream Processing Example, Program Stack.

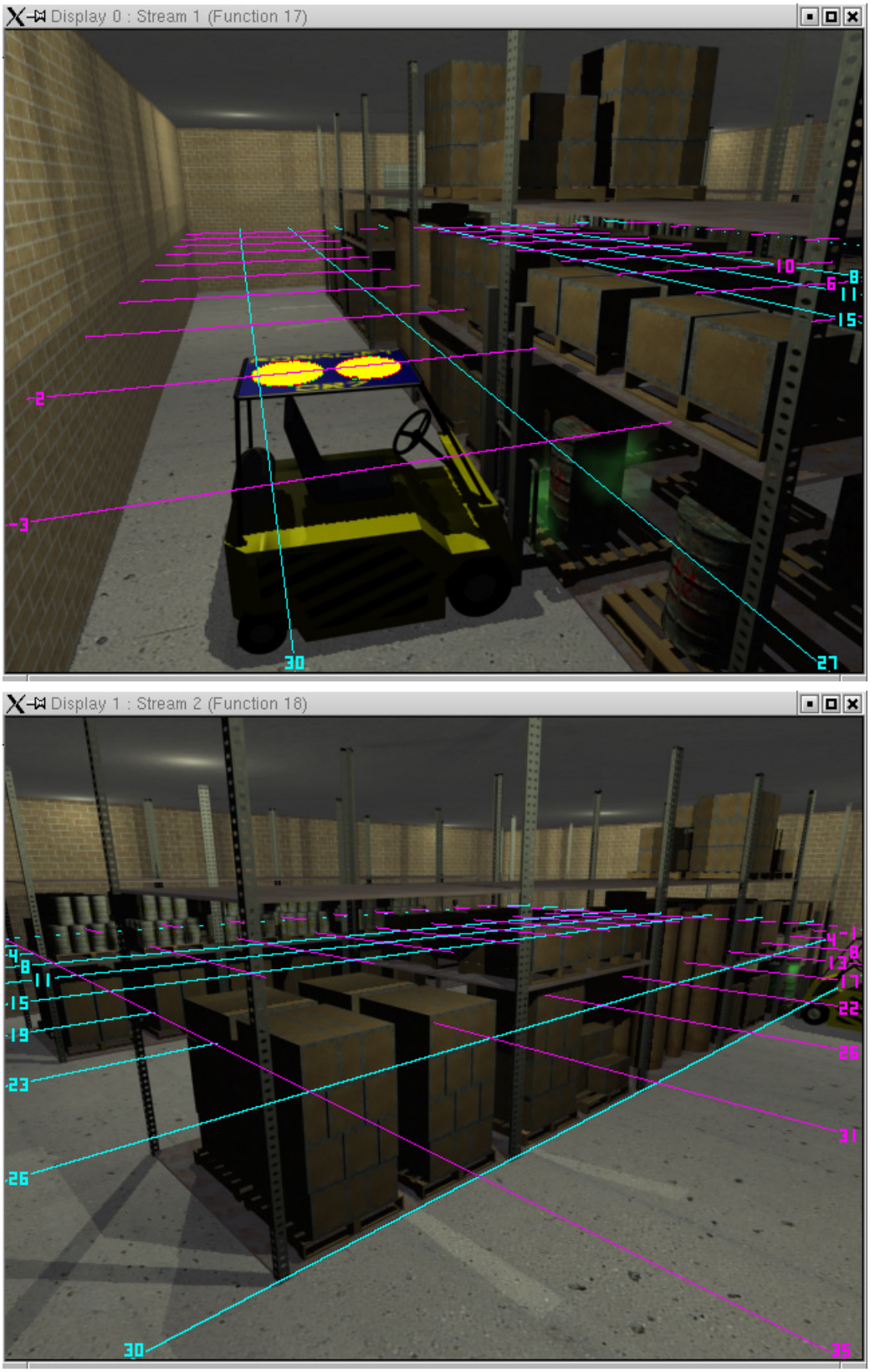

Figure 140: Multiple Stream Processing Example, Video Displays. Stream 1 (Top). Stream 2 (Bottom).

APPENDIX B
TRAINING DATA GENERATION

The process of building training data for the statistical based feature detection algorithms can be a difficult and lengthy process. For this reason, a simple software package was created to automate the process. This process was not considered a significant part of the research, so a minimal amount of time was invested in creating this software. The end result is a command prompt driven program that is thorough and functional, but lacks the 'bells and whistles' that other parts of this research exhibit.

This tutorial explains how to define training data for the type of features discussed in this research, so the information given here is tailored to this specific case of two tracking feature colors on a single background color, see Figure 141. Most of the steps in this tutorial are fairly general, so it should be apparent on how to expand this information to other cases and situations.

Image Editing and Feature Separation

The first order of business is to take a video of the vehicle tracking features with the camera that will be used for tracking. It is necessary to use the intended camera for training data generation because every camera has unique properties that must be accounted for. Furthermore, separate training data should be generated for each camera used in the environment. A series of stills must then be taken from this video to use as training data. This can be done with a video editing utility or though screen captures.

It is advisable to show the vehicle in different positions, lighting conditions, etc. to attempt to capture all possible feature conditions in the training data. Training data

should consist of many pictures, but the actual amount will depend on the situation. Simple environments may only require 5 to 10 training images, while more complicated ones may need 20 or more.

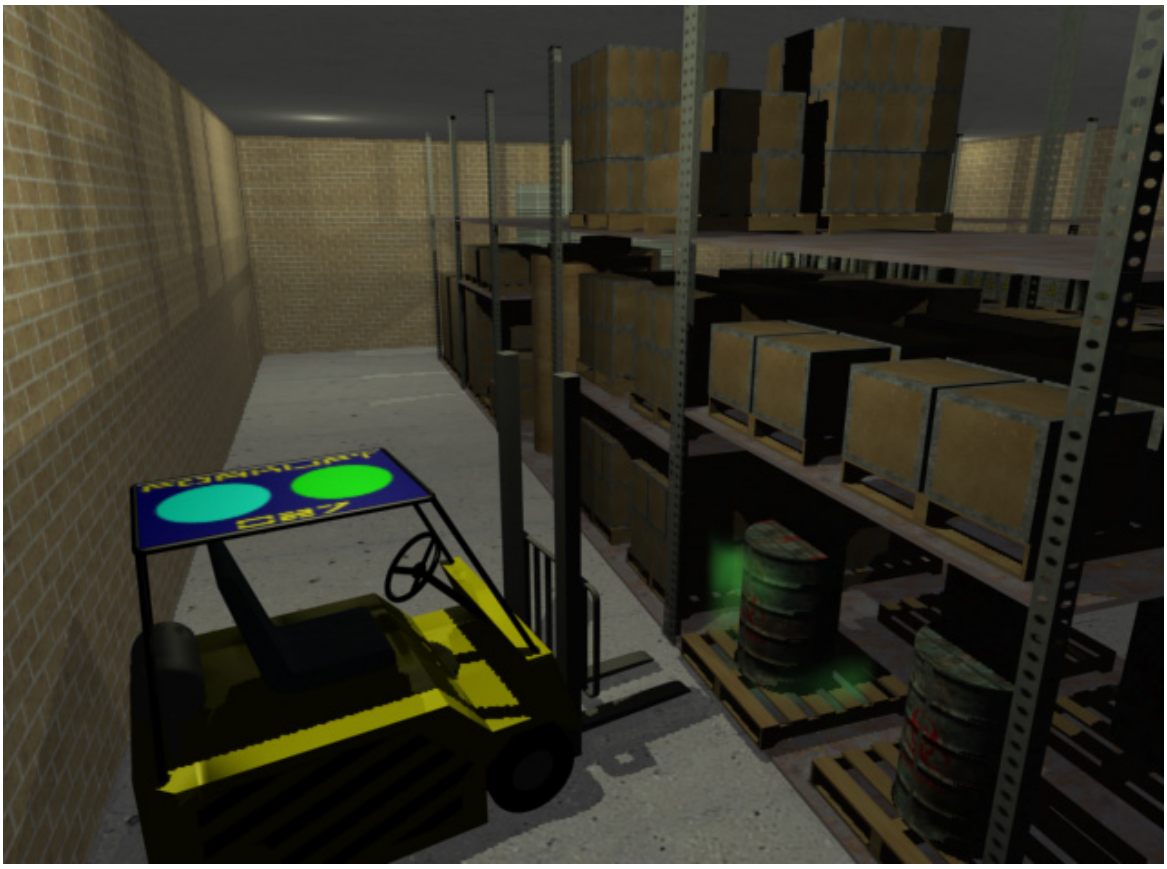

Figure 141: Original Training Data Image

Once the pictures have been taken, the features must be isolated from the remainder of the image. The user must manually find the tracking features in the image and select the corresponding pixels. The author prefers a color range selection tool to do this, but the method does not matter. These highlighted areas are then clipped from the original image and placed in new images. The area left after removing the feature must be full intensity white (255,255,255). This color of white is ignored when the training program is run on the image and therefore will be included in the color distribution of the image. This process must be run separately on the first feature, second feature, and

background, see Figure 142. The new images that are created contain the features and background on white, see Figure 143. Again, full intensity white is ignored when calculating the color distribution.

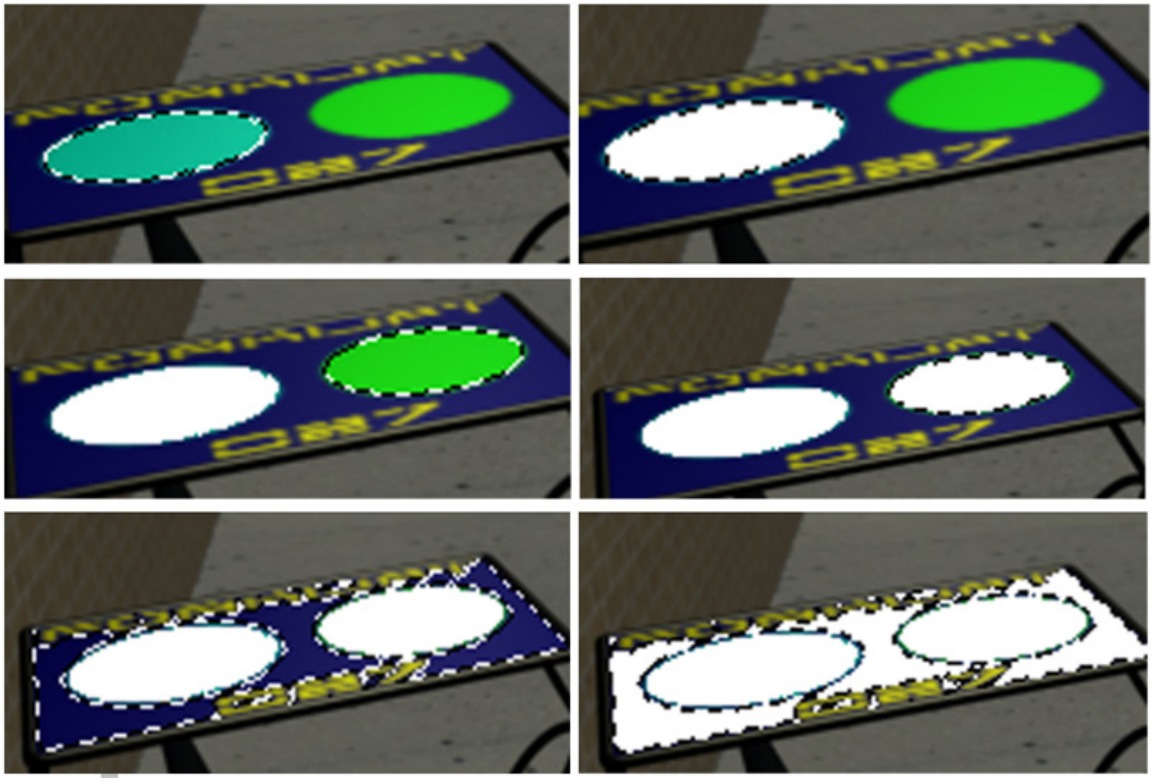

Figure 142: Selection and Removal of Vehicle Tracking Features and Background. Selection and Removal of Cyan Feature (Top Row). Selection and Removal of Green Feature (Middle Row). Selection and Removal of Feature Background (Bottom Row).

Figure 143: New Feature Images. Cyan Feature (Left). Green Feature (Middle). Background (Right).

These images must all be saved as full color bitmaps and placed inside a root folder. This folder contains everything needed to create feature training data: software, data files, and images. Inside the root folder there must be five sub-folders:

‘foreground1’, ‘foreground2’, ‘background1’, ‘background2’, and ‘outside’. Each of the individual images must be saved to the appropriate folder when modified. So the images containing the feature closest to the front of the vehicle (in this case, green) will be saved to the ‘foreground1’ folder. The cyan features, therefore, are saved to the ‘foreground2’ folder. The background will be saved to the ‘background1’ folder. In the case of this research, there will be nothing inside the ‘background2’ folder because only one background was used. Images in the outside folder are used to determine the color distribution of the pixels that are not included as a feature or background, so the original images with the features and background removed are placed in this folder.

The training data generation program requires the user to manually enter the filenames, so it’s beneficial to keep the names simple. The author prefers to use simple names like ‘1.bmp’, ‘2.bmp’, etc. This allows for quick file specification when the training program is run. This process is done for every one of the training data images, so if there are 10 training images there should be 10 images in each folder. This is a fairly lengthy process, but the accuracy and diligence shown in this step will greatly affect the reliability of the position system.

Training Data Generation Program

Now that the images are properly formatted and saved to the proper location, the training data generation program can be run. This searches each image and tallies its color composition. The data from all of the images in each folder are combined and used to create feature extraction classifiers.

In addition to determining the distributions of the training data, the classifiers are run on the training data and error plots are made. Different classifier ranges are cycled

through incrementally and the error is calculated. This data can be used to create graphs in a spreadsheet program to assess the optimal range to use for a data set. The error reduction process can take a very long time, so it may be necessary to run the program overnight if the data set is large.

Starting the Program

The training data generation program is made to run inside of a DOS shell and is command prompt driven. It is, therefore, advisable to have the executable inside the data set's root folder. This way the input files are easily accessible and the output files all stay contained in a known location. When run, the first thing that shows up is the intro screen and the prompt for foreground1 image names, see Figure 144.

```
Image.exe reads data from a sequence of bitmap files to determine
statistical distributions of features in the images.  For program
work properly, there must be five folders inside the root folder:
 * Foreground1   -  has images that contain only the foregound of
                    first feature.
 * Foreground2   -  has images that contain only the foregound of
                    second feature.
 * Background1   -  has images that contain only the backgound of
                    first feature.
 * Background2   -  has images that contain only the backgound of
                    second feature.
 * outside       -  has images with the features removed (i.e.
                    remaining data.
These images must have a consistant background color between them,
typically white (255,255,255).  The root folder must contain the
images with the outside data (i.e. the foreground and background
features removed).  You will need to enter the names of the files
to use manually (without the .bmp) for each portion.
------------------------------------------------------------------


Foreground 1 Images:
 Enter the name of an image to add to the foreground1 table.
 when finished hit (return) twice
>
```

Figure 144: Training Data Generation Program Intro Screen.

The image names without the folder or '.bmp' are typed in, one at a time. The 'Enter' key must be hit after every image name and twice after the final image is input. Once this is done, the program will open all of the images and tabulate their pixel information. The data from this will be saved into the root folder in a file named 'foreground1.dat'. The status of each image is shown as it is processed. Any images

showing ‘Completed’ have been successfully processed. Otherwise, ‘Failed’ will be printed next to the filename, see Figure 145.

```
------------------------------------------------------------------

Foreground 1 Images:
 Enter the name of an image to add to the foreground1 table.
 when finished hit <return> twice
> 1
> 2
> 3
> 4
> Not_an_Image
>

Opening Foreground 1 Input files
 - Processing './Foreground1/1.bmp'...Completed
 - Processing './Foreground1/2.bmp'...Completed
 - Processing './Foreground1/3.bmp'...Completed
 - Processing './Foreground1/4.bmp'...Completed
 - Processing './Foreground1/Not_an_Image.bmp'...Failed


Foreground 2 Images:
 Enter the name of an image to add to the foreground2 table.
 when finished hit <return> twice
>
```

Figure 145: Training Data Generation Program Searching Image

The filenames are entered for both the foregrounds and backgrounds. When ‘Completed’ is printed to the screen, that image’s data has been successfully appended to the output data file for that feature. In this way, the program can be stopped at anytime if the desired information has already been processed.

If the program continues to run, the error checking will commence. This is the time consuming portion of the program, so data is presented at each iteration of the error checking process, see Figure 146. The column headers: MaD, %F1, %F2, %B1, %B2, and %Out relate to the: Mahalanobis Distance, percent error in foreground 1, percent error in foreground 2, percent error in background1, percent error in background 2, and percent error in outside data, respectively. The quantity after the arrow is the total error incurred by all the groups.

```
Foreground 1 Distribution
 MaD   | %F1 | %F2 | %B1 | %B2 | %Out
-------------------------------------------
0.10   | 99.54 | 0.00 | 0.00 | 0.00 | 0.00 -> 99.54
0.20   | 97.97 | 0.00 | 0.00 | 0.00 | 0.00 -> 97.97
0.30   | 93.97 | 0.00 | 0.00 | 0.00 | 0.00 -> 93.97
0.40   | 87.91 | 0.00 | 0.00 | 0.00 | 0.00 -> 87.91
0.50   | 80.27 | 0.00 | 0.00 | 0.00 | 0.00 -> 80.27
0.60   | 71.57 | 0.00 | 0.00 | 0.00 | 0.00 -> 71.57
0.70   | 62.43 | 0.00 | 0.00 | 0.00 | 0.00 -> 62.43
0.80   | 52.84 | 0.00 | 0.00 | 0.00 | 0.00 -> 52.84
0.90   | 43.42 | 0.00 | 0.00 | 0.00 | 0.00 -> 43.42
1.00   | 27.08 | 0.00 | 0.00 | 0.00 | 0.00 -> 27.08
1.10   | 22.33 | 0.00 | 0.00 | 0.00 | 0.00 -> 22.33
1.20   | 20.23 | 0.00 | 0.00 | 0.00 | 0.00 -> 20.23
1.30   | 18.41 | 0.00 | 0.00 | 0.00 | 0.02 -> 18.42
1.40   | 17.05 | 0.00 | 0.00 | 0.00 | 0.02 -> 17.07
1.50   | 16.02 | 0.00 | 0.00 | 0.00 | 0.02 -> 16.03
1.60   | 15.02 | 0.00 | 0.00 | 0.00 | 0.02 -> 15.04
1.70   | 13.74 | 0.00 | 0.00 | 0.00 | 0.02 -> 13.75
1.80   | 12.52 | 0.00 | 0.00 | 0.00 | 0.02 -> 12.54
1.90   | 11.77 | 0.00 | 0.00 | 0.00 | 0.02 -> 11.79
```

Figure 146: Training Data Generation Program Calculating Error

The ‘MaD’ value represents the current value of Mahalanobis Distance that is being used for this classifier. The meaning of the percent error values change depending on the class that is currently being processed. In Figure 146, the foreground 1 distribution is currently being inspected. Therefore, the ‘%F1’ error represents the missed-hit errors of the classifier when searching the foreground 1 images for hits.

At this point every other column represents false-hit errors. The range of Mahalanobis Distances is run on all the image sets to check for accidental hits on non-feature pixels. These error values are then summed up and displayed as total error. It is important to note that the total error does not represent an actual percentage, but a sum of percentages. Because of this it is possible for the total error to exceed 100 percent. This value still conveys the error distribution, just not in relation to a 0 to 100 percent scale.

Output Files

A series of output files are created when the training data generation program runs to give information on various things. The output files are described in detail in the following sections.

Pixel data output files

The program creates five files that contain pixel data information: foreground1.dat, foreground2.dat, background1.dat, background2.dat, and outside.dat. These files contain a list of integers indicating the pixel values of the images in the corresponding folders. The pixel data is in the format of:

Red Grn Blu X Y Image

Where *Red*, *Grn*, and *Blu* represent the red, green, and blue integer values of a pixel in the image. Every pixel in every image in the folder is represented in this fashion and, each row represents a single pixel. The pixels that are full intensity white are considered to be empty space and, therefore, are not displayed in the file. *X* and *Y* give the image coordinates of each pixel. These values are present so that any algorithms that need the relative locations of the pixels have access to that information. The final value, *Image*, tells the image that the pixel came from. This value correlates to the order that the images were processed. Therefore, pixels coming from the first image processed will have an *Image* value of 0, pixels coming from the second will have a 1, and so on. The information relating the *Image* value to a filename can be found in the stats.txt file.

Tracking data file

The 'output.trk' file is the program generated tracking file that can be used with the position system software. This file lists the classifier results from the program, as well as, the optimal range values from the error reduction stage. This information is listed in a fashion that is easily read and understood by the position system software, but is still ASCII based so that it can be modified by the user outside of the program. The file lists five characters to identify the variable and then the value of that variable. The first character must be a '1' or '2' to indicate feature one or two. The second character

must be an ‘f’ or ‘b’ to indicate foreground or background. The following three characters are the operational codes that relate to the specific variable for that feature. The identifiers are listed in Table 20.

Table 20: Tracking File Op-Codes

ID String	Membership	Data Type	Description
rlo	Color Range	Integer	Low pixel value of red
rhi	Color Range	Integer	High pixel value of red
glo	Color Range	Integer	Low pixel value of green
ghi	Color Range	Integer	High pixel value of green
blo	Color Range	Integer	Low pixel value of blue
bhi	Color Range	Integer	High pixel value of blue
rdi	Color Direction	Double	Red color vector direction
gdi	Color Direction	Double	Green color vector direction
bdi	Color Direction	Double	Blue color vector direction
rva	Color Direction	Double	Variance of classifier in the red direction
gva	Color Direction	Double	Variance of classifier in the green direction
bva	Color Direction	Double	Variance of classifier in the blue direction
rme	3D Gaussian	Double	Mean of the classifier in the red direction
gme	3D Gaussian	Double	Mean of the classifier in the green direction
bme	3D Gaussian	Double	Mean of the classifier in the blue direction
irr	3D Gaussian	Double	Red-red value of the inverted covariance matrix
igg	3D Gaussian	Double	Green-green value of the inverted covariance matrix
ibb	3D Gaussian	Double	Blue-blue value of the inverted covariance matrix

Table 20. Continued

ID String	Membership	Data Type	Description
irg	3D Gaussian	Double	Red-green value of the inverted covariance matrix
irb	3D Gaussian	Double	Red-blue value of the inverted covariance matrix
igb	3D Gaussian	Double	Green-blue value of the inverted covariance matrix
3dd	3D Gaussian	Double	Mahalanobis Distance to use with the classifier
xme	2D Gaussian	Double	Mean of the classifier in the X direction
yme	2D Gaussian	Double	Mean of the classifier in the Y direction
ixx	2D Gaussian	Double	X-X value of the inverted covariance matrix
ixy	2D Gaussian	Double	X-Y value of the inverted covariance matrix
iyy	2D Gaussian	Double	Y-Y value of the inverted covariance matrix
2dd	2D Gaussian	Double	Mahalanobis Distance to use with the classifier
met	All	Hex *(Integer)*	Specifies which classifier to use: 0x00 – 3D Gaussian 0x01 – 2D Normalized Gaussian 0x02 – Color Direction 0x03 – Color Range

This is the general information on how to use the tracking file. More specific and detailed information can be found in Chapter 5.

Error values file

The error values calculated in the training data generation program are saved to the 'output.csv' file. This file is formatted in such a way that it is easily imported into Microsoft® Excel or any other spreadsheet program. A spreadsheet program will allow the data to be plotted in such a way that the error trends are easily recognized so that the proper range values can be used.

This file has an extension of '.csv' to indicate to spreadsheet programs that it is a comma separated values file. This generic format contains no software specific information, so it is easily recognized by any spreadsheet package. The data in this file is comprehensive, allowing a user to make informed decisions about the classifier from this file alone.

The data in this file starts with the classifier specific information about each type of classifier, including the range values, mean values, etc. This information is listed on a line-by-line basis that can be easily reviewed. The data in each line is separated by commas in the following fashion:

Range,%F1,%F2,%B1,%B2,%Out

The Range descriptor is used here in stead of the MaD shown in Figure 146. This range value is different for each type of classifier. For the two Gaussian based classifiers the Range variable will represent the Mahalanobis Distance. For the other two classifier, this value will represent various color ranges for each color channel. Regardless of the type of classifier, the error plot should resemble the plot shown in Figure 147.

Program status file

The 'stats.txt' file is a program status file that can aid if any problems occur while running the program. This file contains information about which images were processed in which sections. The images that are inspected in each folder are listed along with their associated image ID number, image resolution, total number of pixels, number of pixels kept from that image when processed, and the number of pixels ignored. This information allows the user to see which images were processed properly and how each image contributes to the overall distribution for that feature.

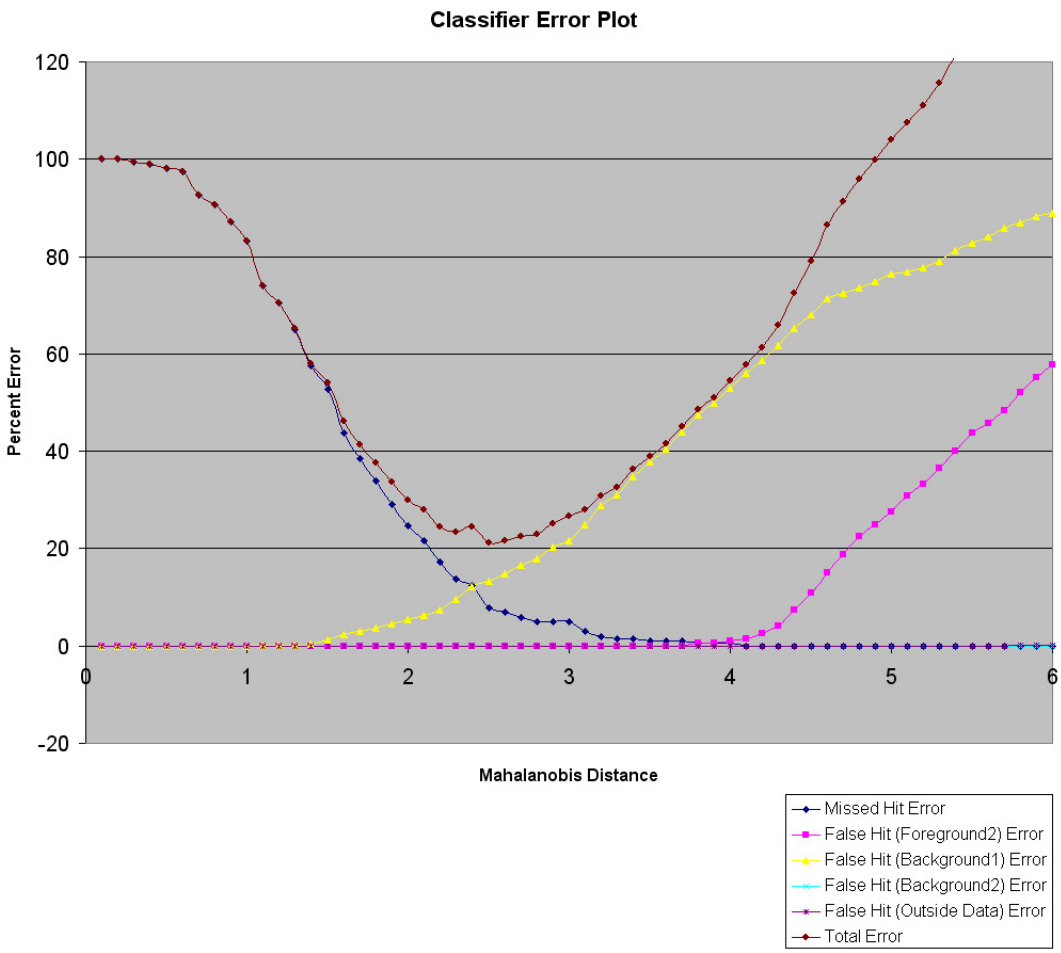

Figure 147: Sample Error Plot

APPENDIX C
IMAGE DATA RE-SAMPLING

When using numerous images for classification, the data sets can range into the tens of millions of data points. This amount of information is excellent for determining statistical information about a system, but can be problematic when it is necessary to display this in graphs and charts. Some of the mathematic packages that were used to generate graphs for this document can only handle hundreds of thousands of points, not millions. Even with a few hundred thousand data points, the calculations are lengthy and slow.

Visualizing data distributions can be a powerful tool when trying to determine a classifier, so down-sampling can be used to condense an oversized data set to a tolerable range. A data set's distribution is defined by its full compliment of data points, but some information is lost when the number of data points is reduced. Converting very large data sets (over a million points) to an acceptable range (around a hundred thousand) can result in some degradation of the original distribution, but this is acceptable for the sake of visualization.

A data re-sampling program was created to handle the downsizing of image data. This program determines the initial distribution of the large set of data and attempts to recreate the same distribution with a smaller sample size. While, this program can re-sample a data set to almost any size, care must be taken to not destroy the data distribution in the process.

Re-Sampling Data

The process of taking new data samples from the same distribution is referred to as data re-sampling. Data re-sampling comes in two different varieties: down-sampling and up-sampling. Data down-sampling entails reducing the number of data points in a distribution. Data can also be up-sampled so that the same distribution is defined by more data points. Both of these types of data re-sampling can be useful in certain situations.

Determining Data Distribution

The first aspect of data re-sampling, for down or up-sampling, is to determine the initial data distribution. For the purpose of this research, the data is assumed to be in terms of red, green, and blue values. This particular program only re-samples data files from the training data generation program. This requires that the data files be in the format:

Red, Grn, Blu, X, Y, Image

Where *Red*, *Grn*, and *Blu* are the integer pixel values of the red, green, and blue color channels, respectively. *X* and *Y* are the image coordinates of that pixel in the original image and, *Image* is the image ID for the picture that the pixels came from. More information about the data generation program can be found in Appendix B.

The image data is sorted and placed in a 256×256×256 array. This array holds an integer tally for every one of the 16,777,216 possible color values in a full color image. The array element corresponding to a particular color is incremented by 1 every time that color is found in the input file. The final result is an array that has a tally of all the image pixels.

Creating a New Data Set

Tallying the data is the quick portion of the re-sampling process. Creating the new data set takes much more time. The new data set is created by multiplying the values in every element of the array by a scaling factor. This factor is given by:

$$SF = \frac{N_{desired}}{N_{actual}} \tag{119}$$

Where SF is the scaling factor, $N_{desired}$ is the number of data points that are desired and N_{actual} is the number of data points in the original set.

When the scaling factor is multiplied by each element in the array, a floating-point value is created. Obviously, a partial data point cannot be displayed, so the value must be rounded to an integer value. Unfortunately, the floating-point value cannot simply be rounded using the half-way rule. Since many of the array elements are likely to have only one pixel of a particular color, it is insufficient to round the floating-point value at the half way mark. This will cause all of these pixels to be changed to zero and therefore, greatly degrade the distribution.

Processing in this fashion creates a new distribution that greatly favors colors with a high pixel count and ignores colors with smaller pixel counts. While this barely changes the statistical properties of the distribution, the graphs and charts showing data from the down-sampled distributions look segmented and unrealistic. This problem only occurs on down-sampled data sets. To compensate for this, a different method of rounding data points is used. Instead of rounding at the half-way point, rounding is done at the fractional value of the scaling factor. For instance, if a data set is being scaled down to 10% of its original size, the rounding point would be 0.1. Any floating point

values smaller than 0.1 are rounded to zero and values higher are rounded to 1. This way, every color in the distribution is preserved.

Data Re-sampling Program

The data re-sampling program is a simple DOS based command prompt program that takes a properly formatted data file and re-samples the data at a rate specified by the user. This process is not perfect, and data set sizes rarely come out to the exact sample size desired, but the sizes should be fairly close to the user-specified value.

When run, the program will prompt the user for the name of the file to re-sample. When entered, the program scans the file and tells the user how many data points are currently in the file. It then prompts the user for the desired size of the new sample set. Once the user inputs this information, the program processes the set and saves the new data file to the same filename with a 're_' prefix. So for the data file 'roadlines.dat' the file is renamed to 're_roadlines.dat'. Once the processing is complete, the program tells the user how many data points are present in the new data set, see Figure 148.

```
Enter the name of the file to resample
> roadlines.dat

Scanning Input File...Done
roadlines.dat currently has 1093 data points. What would you like to resample
the data to?
> 20000

Resampling Image...Done
Saving Output File...Done

A total of 19996 data points were written
Press any key to continue . . .
```

Figure 148: Data Re-sampling Program

Testing New Data Sets

Of course, the new data sets need to be accurate in their representation of data, so a number of tests were performed on up-sampling and down-sampling data. To test the changing data sets, a plot was made of the original data. Then the data set was re-

sampled at a few different rates. The graphs from each of these re-sample rates are shown in comparison to the original to see the effects of the re-sample.

Down-Sampling Tests

Down sampling is the more important of the re-sampling types so it was the most scrutinized. The non-feature data from the simulated warehouse example was used for this test. Since non-feature data consists of all the pixels that are not considered to be features or backgrounds, the data sets are typically very large and need to be down-sampled.

The non-feature data used for this test comes from the four training images used for the warehouse. These images can be found in Chapter 3. The distribution for this data falls along two major paths in RGB space with considerable scattering along these directions. Down-sampling data too much can cause the scattered portions of the distribution to disappear.

The warehouse non-feature data is shown in Figure 149 on page 262 with original distribution compared to four different down-sampling cases. The original data set consists of 1,217,769 data points. In the down-sampled images, the sample size is reduced to: 515,457 for the first set, 108,206 for the second set, 56,763 for the third set, and 12,853 for the fourth set.

In this figure it can be seen how the distribution degrades as the sample size decreases. The first set with 515,457 points looks almost identical to the original distribution. The results are practically the same with the 108,206 point set. A few spikes can be seen in the distribution, but the degradation is minimal. When the sample size is reduced to 56,763, the noise that occurs in the previous set is more apparent. The final set, containing only 12,853 data points, shows significant spiking along the

distribution curve. The general shape of the distribution is the only thing that remains the same.

While the down-sampled distribution problems can be seen fairly well in the color histograms, it is even more visible in the RGB space plots, see Figure 150 on page 263. In these plots the data that is being removed becomes very apparent. In the first down-sampled case, the distribution appears identical, but the point cloud is slightly thinner. This indicates that there are less data points in this new distribution. The second case shows significant degradation in the distribution. One leg of the distribution is much thinner and almost appears to separate from the main portion of the distribution. The third case actually separates the two clouds, making it look like there are two separate distributions. The final case, where the new distribution is the smallest, the degradation to the original data is so severe that the RGB space representation does not even appear to be related to the original case.

Up Sampling Tests

Up-sampling data is not as critical as down-sampling, but tests were still done to ensure the integrity of the up-sampled data. Up-sampling causes practically no changes to the original distribution because there is no risk of a color's representation reducing to zero. Slight changes can occur due to inconsistencies caused by rounding, but these differences are practically transparent.

For the up-sampling tests, the feature data from the warehouse samples was used. This data set was extremely small for image data, only having 11,031 data points. Comparing the histograms from the feature and non-feature data sets require that they have the same number of data points, or else the comparison is meaningless. The non-feature data has been down-sampled, so the feature data must be up-sampled.

Since it was determined that the data does not significantly degrade when up-sampled, only two test cases are shown. The first case practically doubles the data set size, raising the new size to 21,716 data points. The second case increases the set size by nearly fifty times, giving the new set 498,034 data points. The results for both cases are indistinguishable from the original data set. The histogram comparison is shown in Figure 151 on page 264 and the RGB space comparison is shown in Figure 152 on page 265. The histograms and the RGB space plot show that the data is, for all practical purposes, identical in all three cases.

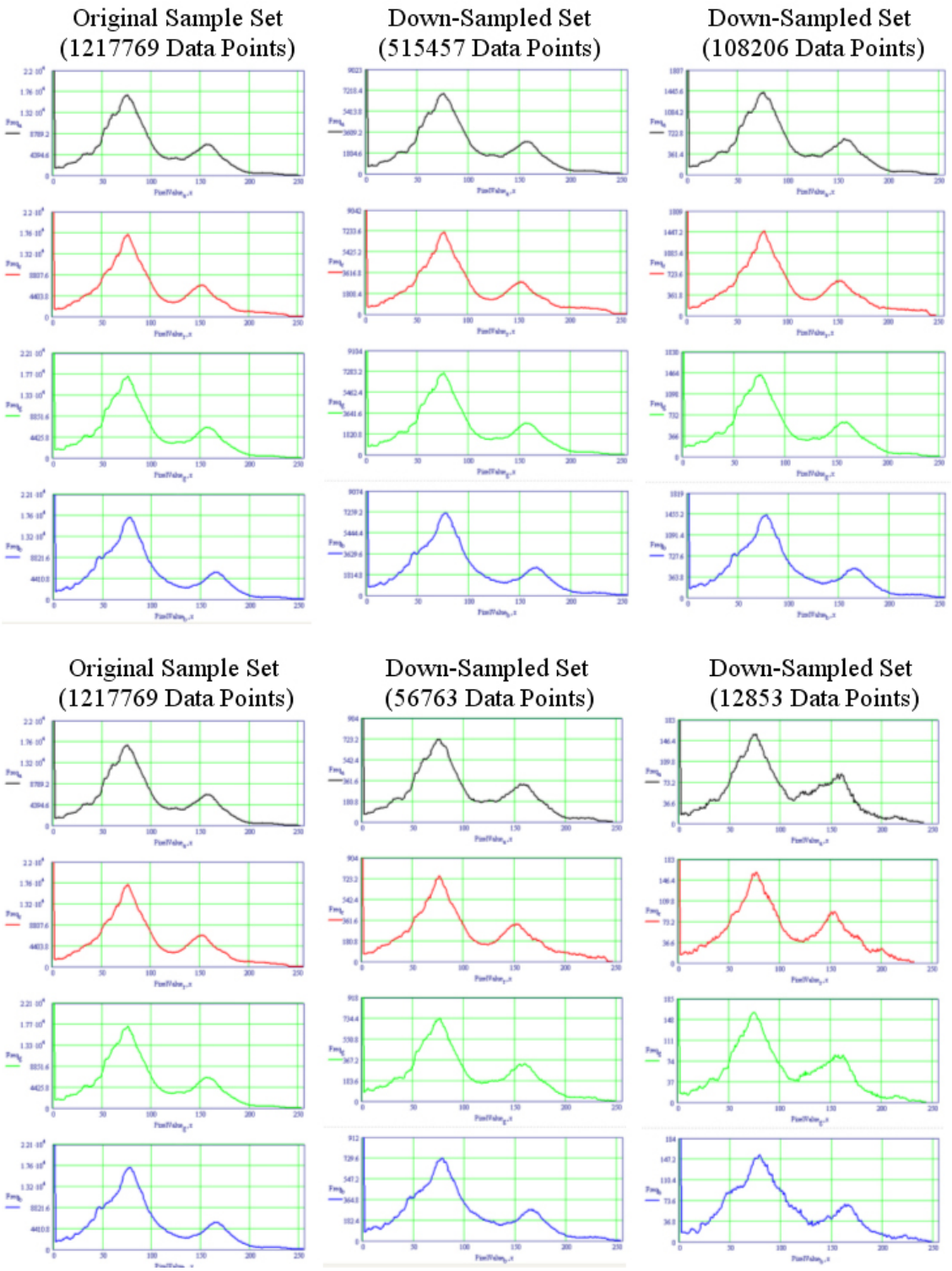

Figure 149: Original Distribution vs. Down-Sampled Distributions

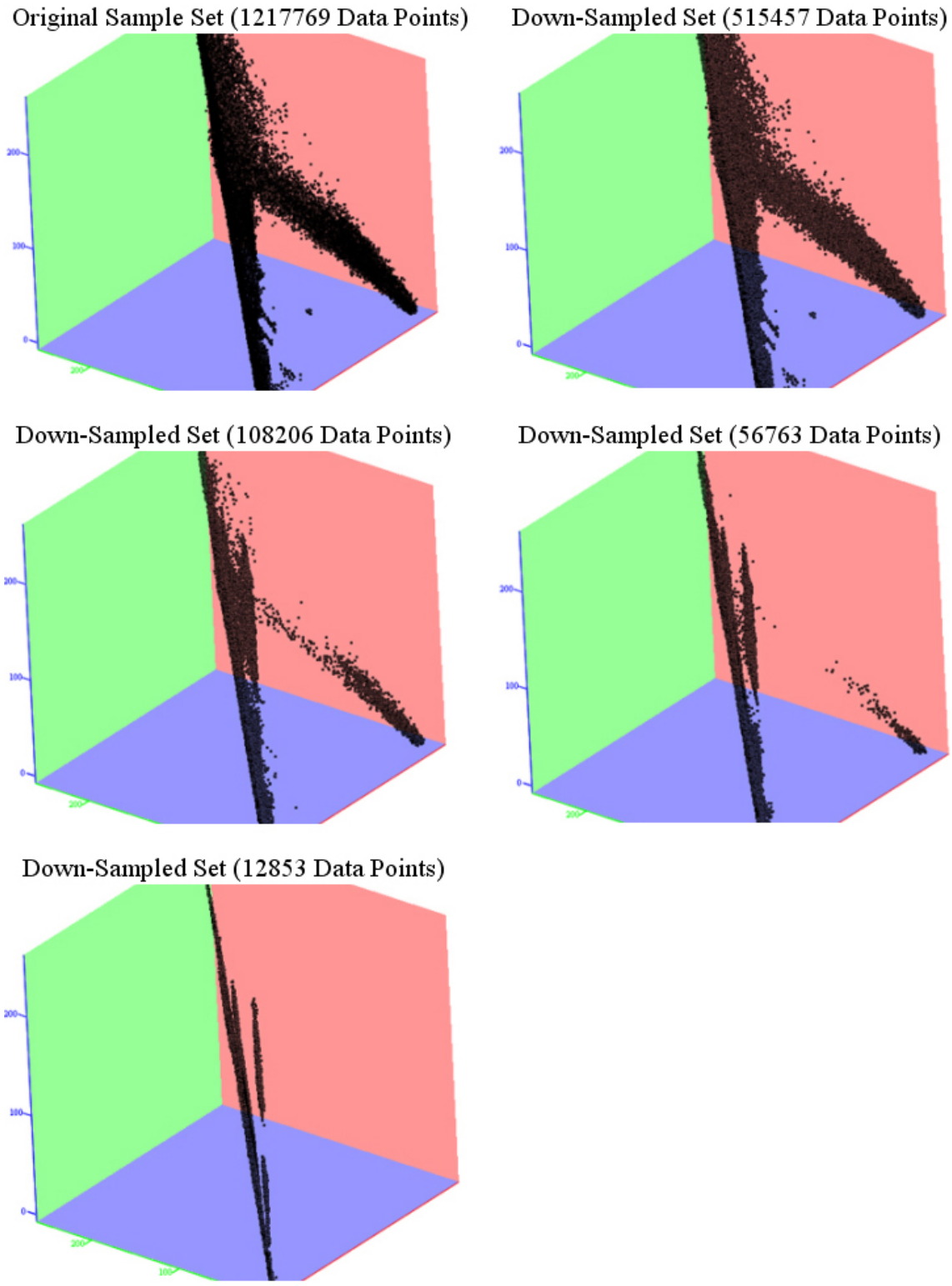

Figure 150: Original Distribution vs. Down-Sampled Distributions in RGB Space

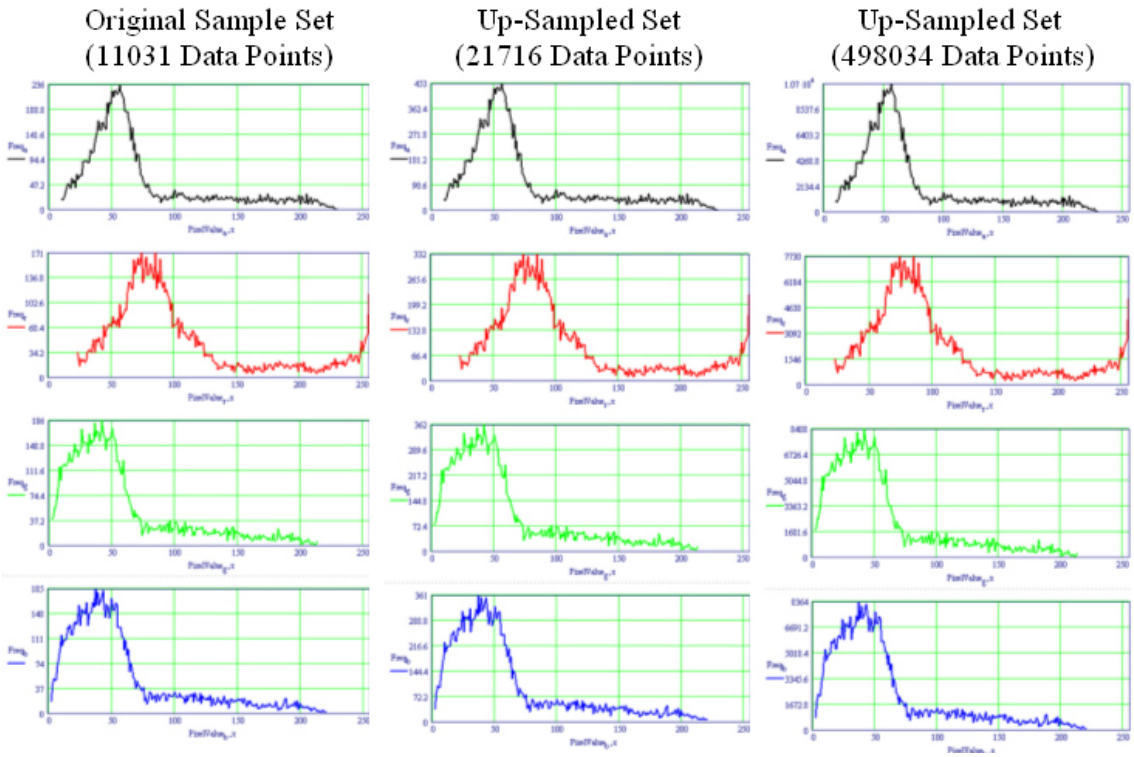

Figure 151: Original Distribution vs. Up-Sampled Distributions

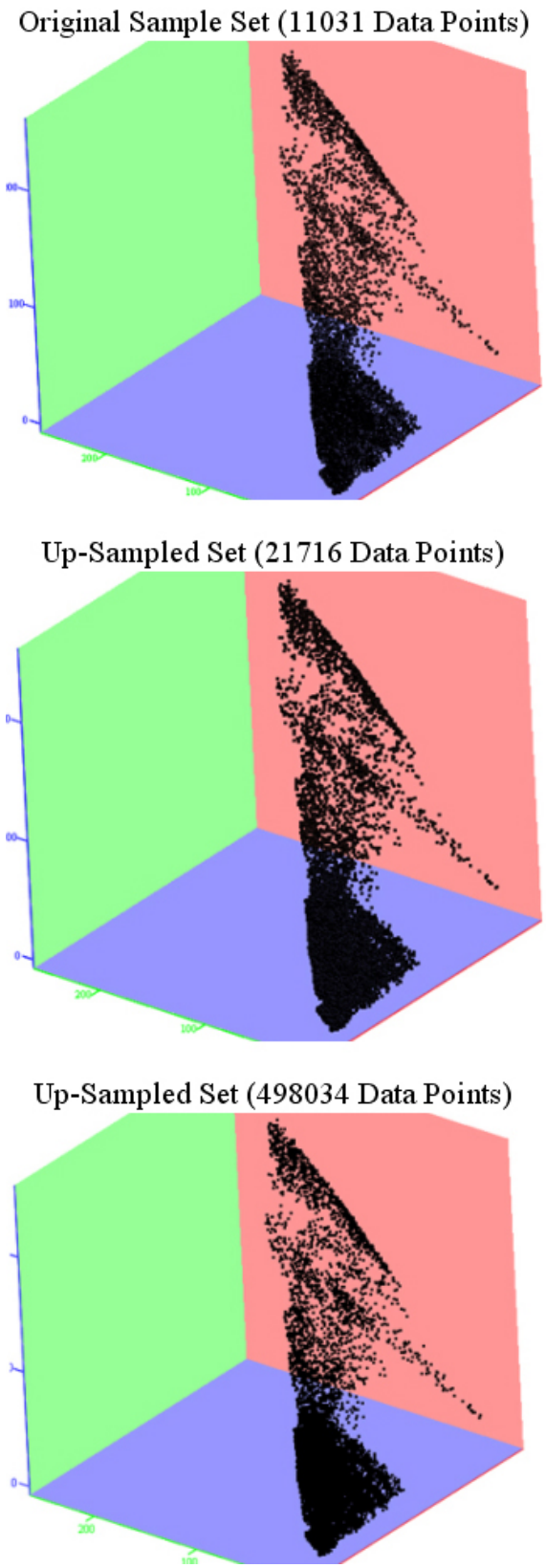

Figure 152: Original Distribution vs. Up-Sampled Distributions in RGB Space

APPENDIX D
IMAGE DATA SPLITTING

To see the effect a classifier has on a data set, the results of the classifier can be shown as four different sets of points: correct hits, correct misses, missed hits, and false hits. To get these data sets, the classifier must be run on the feature and non-feature data sets. Any pixels the classifier finds in the feature set are considered to be 'correct hits', while any pixels that the classifier misses are labeled as 'missed hits'. Pixels found by the classifier in the non-feature data set are labeled as 'false hits' and the pixels missed are 'correct misses'. These four data sets can be graphed in separate colors to display the results of the classifier.

```
Enter the feature input file
> foreground1.dat

Enter the outside data input file
> outside.dat

Which type of classifier should be used?
 1) Color Range
 2) Color Direction
 3) 3d Gaussian
 4) 2d Normalized Gaussian
_
```

Figure 153: Data Splitting Program Entering Initial Information.

The data splitting program automates the process of splitting two pixel data files into four data sets. The data splitting program starts out by asking for the filenames of the feature and non-feature data files. If using data from the training data generation program, the typical filenames are going to be: foreground1.dat, foreground2.dat, and outside.dat. Once the names have been entered, the classifier type must be chosen, see Figure 153. The data splitting program can process data with any of classifier types defined in this document.

Once the classifier type is chosen, the specific information for that classifier must be entered. Of course, the classifier variables are different for each type of distribution. For some classifiers there is more than one way to enter the information, in which case, the program will prompt for which type of information will be given, see Figure 154.

In this case, the 3D Gaussian, the mean and covariance matrix information must be given to define the classifier. Since certain portions of this research deal with the covariance matrix and others deal with the inverted covariance matrix, the program allows both types of entries.

```
 4) 2d Normalized Gaussian
3

Enter the classifier data:
Mean R > 128.3
Mean G > 212.5
Mean B > 235.7

Would you like to enter the Sigma matrix
or the Inverted Sigma Matrix (s/i)> i

InvRR > 1231.2
InvGG > 3241.2
InvBB > 234.6
InvRG > 124.6
InvRB > 975.5
InvGB > 231.1
Mahalanobis Distance > 3.5
Press any key to continue . . .
```

Figure 154: Data Splitting Program Entering Classifier Information

Once all of the information has been entered into the data splitting program, the input files are processed. Data from both files is processed with the user-defined classifier. All the pixel information is put in one of four files: CorrectHit.dat, CorrectMiss.dat, FalseHit.dat, or MissedHit.dat.

The feature data file is the first to be processed. If the classifier determines that a point in this file is a feature, that point is copied into the CorrectHit.dat file. Any pixels in this file that are missed by the classifier are copied into the MissedHit.dat file. Once the feature file is complete, the non-feature file is checked. The same procedure is

employed, but any pixels found by the classifier in this file are copied into the FalseHit.dat file. Any pixels that are not determined to be features by the classifier are copied into the CorrectMiss.dat file.

Once the processing is completed, all of the data from the two input files is distributed throughout the four output files. The data in these files is in the format:

Red, Grn, Blu

Where *Red, Grn,* and *Blu* indicate the red, green, and blue values of each pixel, respectively. This data can now be imported into any desired software package. The graphical results of data splitting can be seen in Figure 155. This figure shows the original data set graphed in RGB space on the left. The white points are feature pixels and the black points are non-feature pixels. The right graph shows the same data split up by a classifier. The white pixels are now the ‘correct hits’, the black pixels are ‘correct misses’, the blue pixels are ‘missed hits’, and the red pixels are ‘false hits’

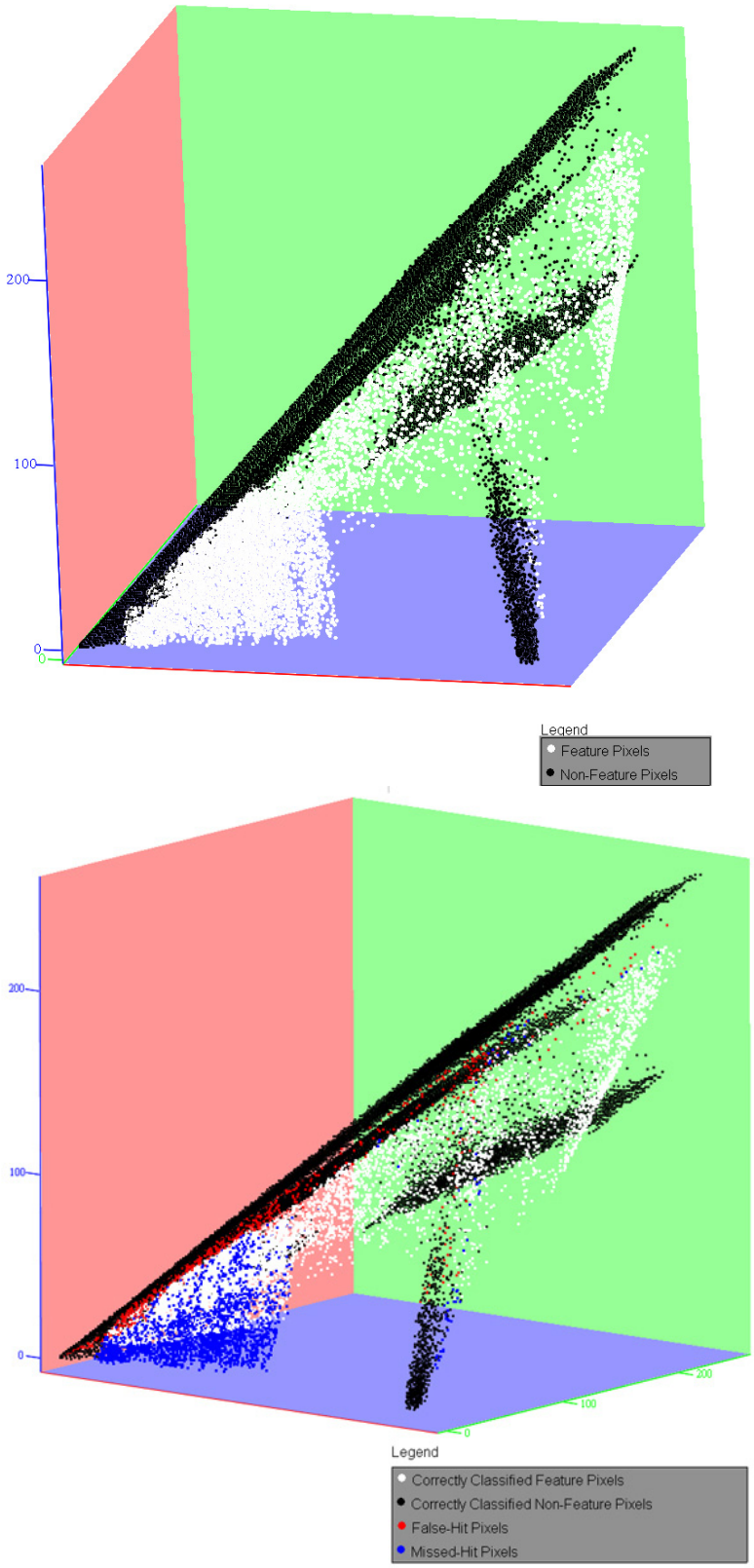

Figure 155: Graphical Results of Data Splitting. Input Data (Top). Output Data (Bottom)

LIST OF REFERENCES

Alo97 Aloimonos, Yiannis, *Visual Navigation: From Biological Systems to Unmanned Ground Vehicles*. Lawrence Erlbaum Associates, New Jersey, 1997.

Bra92 Brain, Marshall., *Motif Programming: The Essentials . . . and More*. Digital Press, 1992.

Cam94 Cameron, S., Probert P., *Advanced Guided Vehicles: Aspects of the Oxford AGV Project.* World Scientific Publishing Co., 1994.

Cro98 Crow, Robert P., "Federal Radionavigation Plan – Pie in the Sky of Civil Aviation?" *Proceedings from the 1998 IEEE Position Location and Navigation Symposium*, p.278-294. Institute of Electrical and Electronics Engineers, New York, 1998.

Del02 Delphi Graphics File Formats and Conversions Page. http://plum.ia.polsl.gliwice.pl/~DIP/efg/Library/Delphi/Graphics/FileFormatsAndConversion.htm (accessed June 2002).

Dud01 Duda, Richard O., Hart, Peter E., and Stork, David G., *Pattern Classification*. 2nd ed. John Wiley & Sons, New York, 2001.

FAS03 Federation of American Scientists Home Page. http://www.fas.org/ (accessed March 2003).

Lin01 Lin, Norman, *Linux 3D Graphics Programming*. Wordware Publishing, Inc., Texas, 2001.

Man01 Mandepat, Rommel E. "Development and Evaluation of Positioning Systems for Autonomous Vehicle Navigation" Master's Thesis, University of Florida, 2001.

McG03 McGowan , John F., Ph.D. Home Page. http://www.jmcgowan.com/ (accessed March 2003).

MDN02 Microsoft Developer's Network Home Page. http://msdn.microsoft.com (accessed March 2002).

Ndi98 Ndili, A., Enge P., "GPS Reciever Autonomous Interference Detection" *Proceedings from the 1998 IEEE Position Location and Navigation Symposium,* p. 315-331. Institute of Electrical and Electronics Engineers, New York, 1998.

Pos02 Poskanzer , Jef. Ohio State University, Computer Information Sciences Class Pages. http://www.cis.ohio-state.edu/~parent/classes/681/ppm/ppm-man.html (accessed June 2002).

Tri03 Trimble, Paula S., Federal Computer Week Online. http://www.fcw.com/ (accessed March 2002).

Vin00 Vincze, Marcus, Hager, Gregory D., *Robust Vision for Vision-Based Control of Motion*. IEEE Press, New York, 2000.

Wat00 Watt, Alan, *3D Computer Graphics*. 3rd ed. Addison-Wesley, London, 2000.

Wer94 Wernecke, J., *The Inventor Mentor: Programming Object-Oriented 3D Graphics with Open Inventor*^{TM}*, Release 2*. Addison-Wesley, New York 1994.

Wit96 Wit, Jeff, "Integrated Inertial Navigation System and Global Positioning System for the Navigation of an Autonomous Ground Vehicle," Master's Thesis, University of Florida, 1996.

Yel03 Yeluri, Sai, "Outdoor Localization Technique Using Landmarks to Determine Position and Orientation," Master's Thesis, University of Florida, 2003

BIOGRAPHICAL SKETCH

Anthony Hinson was born in Lakeland, Florida, a small town northeast of Tampa in 1976. As a child he was fond of computers and would spend hours writing programs that rarely worked. By adolescence, his interests had turned to automobiles and electronics, and he could often be found under the hood or behind the dash of a car. These interests eventually guided him into the field of engineering.

Directly after high school, Anthony started his bachelor's in mechanical engineering at the University of South Florida (USF). During this time, he was an officer and member of several different engineering societies including ASME, AIAA, Pi Tau Sigma (a mechanical engineering honor society), and Tau Beta Pi (a multi-discipline engineering honor society).

During his final years at USF, Anthony had the privilege of working as an intern in Honeywell's Space and Strategic Systems Operations (SASSO) division. Here, Anthony worked with a team of engineers on defense related contracts while learning the differences between scholastic and applied engineering. While at Honeywell, Anthony also had the opportunity to work with local high school students to design and fabricate a robot for the F.I.R.S.T. competition. Anthony led the graphical design and animation team.

Anthony graduated from the University of South Florida with high honors in December of 1999, and decided to continue his education with a master's degree from the University of Florida (UF). Throughout his time at UF, Anthony was a research assistant

in the Center for Intelligent Machines and Robotics (CIMAR) and worked with other students on research related topics. Anthony's interests turned to the computer vision area of robotics and his research soon followed. Anthony completed his master's degree in the summer of 2003 and is currently seeking work in the field of robotics and computer vision.