Implementing a Real-Time Hough Transform on a Mobile Robot

An Honors Paper for the Department of Computer Science

By John Gordon Morrison

Bowdoin College, 2011

© 2011 John Gordon Morrison

Contents

List of Figures **v**

Acknowledgements **vii**

Abstract **vii**

1 Introduction **1**
1.1 Related Work 3
1.2 Overview 6

2 Edge Detection **9**
2.1 Sobel Operator 9
2.2 Edge Peak Detection 11

3 Hough Transform **15**
3.1 Voting 16
3.2 Peak Detection 20
3.3 Line Candidate Narrowing 20

4 Field Line Detection **21**

5 Results **23**
5.1 Edge Detection Performance 25
5.2 Hough Transform Performance 27

6 Future Work **31**

7 Conclusion **33**

Glossary **35**

Appendices **39**

A Code **39**

B Image Test Sets **41**

References **41**

List of Figures

1 Aldebaran Nao 1
2 Confusing off-field objects 2
3 Two images from different venues, color segmented with the same table 4
4 Image Processing Pipeline 7
5 Reducing edges with the field horizon 11
6 Asymmetric Peak Test 12
7 Parallel lines with opposite directions 15
8 Example of lines passing through a point 16
9 Hough space marked with full angle 17
10 Example of one edge peak's votes in Hough space 18
11 A line obstructed by a robot 21
12 New line detection does not depend on color calibration 23
13 Average Full Image Line Detection Run Time 24
14 Average Sobel Operator Run Times 25
15 Average Edge Peak Detection Run Time 27
16 Average Hough Voting Run Time 27
17 Average Hough Space Smoothing Run Time 29

Acknowledgements

Bill Silver has been a tremendous resource and a driving inspiration behind this project. His guidance and instruction were invaluable.

The advising and support of student research from Professor Eric Chown was always encouraging and helped keep me on track throughout the project.

Abstract

Robotic vision is a challenging problem due to the uncertain nature of real-world environments and the computational constraints of mobile platforms. Many computer vision algorithms have high computational requirements and are often seen as unsuitable for use in an embedded system. RoboCup, an international robotic soccer competition, presents a dynamic and mobile environment in which color based run-length encoding algorithms struggle. Many current approaches use static and fragile color lookup tables and heavily subsampled images to achieve an optimal processing rate of 30 frames per second.

The aim of this project is to implement the Hough Transform for real-time line finding on the Aldebaran Nao robot. Using mostly color and lighting independent pixel gradient information, we detect soccer field lines. Using assembly language instructions to expose the processor's full potential, this project implements a real-time Hough transform based object detection system on 320x240 pixel images.

1 Introduction

Computer vision and image processing are commonly applied in desktop software applications, such as Apple's iPhoto [12], and in manufacturing for such diverse tasks as quality control, robot guidance, assembly verification, and code reading. [6] Embedded mobile robot vision is progressing, as well. [8] However, vision is often not the first choice for perception for mobile robots. For instance, the winning entry to the DARPA Grand Challenge in 2005 used only minimal vision processing, instead relying on laser scanners for the majority of its perception [7]. Advanced, high resolution computer vision is computationally intensive and very complex, which can restrict its use in mobile environments.

The **Standard Platform League (SPL)** [15], an autonomous robotic soccer competition, in contrast, uses robots whose most accurate and primary source of perceptual information is their camera, forcing real-time vision processing to become a topic of intense research and development. The SPL uses a standardized robot built by a third-party company. No team is allowed to modify its robot in any way, so the competition is based purely upon software.

The SPL environment and platform offer a particular set of constraints which challenge any potential vision system, such as a relatively slow (by desktop standards) processor, a narrow field of view, unpredictable images, partially obscured objects, and limited processing time per frame. The output of the vision system, usually in the form of a distance and bearing to a sighted field object, is the foundation for higher level cognition and skilled soccer playing.

1.0.1 Platform

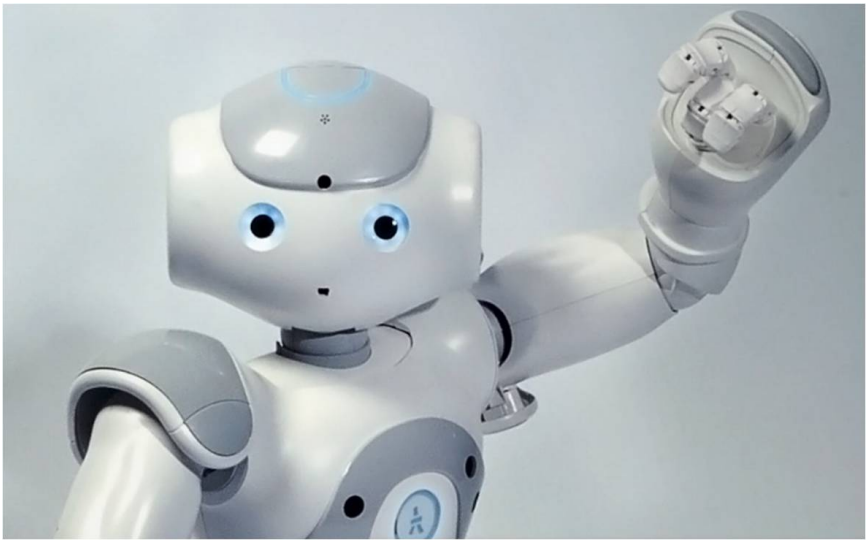


Figure 1: Aldebaran Nao robot [19]

The SPL uses the Nao humanoid robotic platform, manufactured by Aldebaran. The Nao comes equipped with a 500 MHz AMD Geode LX processor which, in

addition to the standard x86 instruction set, supports the Intel **MMX** and AMD **3DNow! Single Instruction Multiple Data (SIMD)** instruction sets. The Nao camera can capture a 640x480 pixel image at up to 30 frames per second and stores it in the **YUV** 422 format.

YUV pixels are composed of three channels: Y, U and V. The Y channel carries the luma, or brightness, of the pixel. Without the U and V channels, the Y represents a black and white image. The U and V channels deliver the chroma, or color, of each pixel. [14]

The 422 label specifies that for every 4 bytes of Y information, there are 2 bytes each of U and V information, giving the brightness twice the resolution compared to the color. The bytes in the Nao's image are aligned as Y_1UY_2V. Note that each pair of Y_1 and Y_2 share a single UV pair. The camera can also provide a 320x240 YUV422 image, if desired. [19]

The camera has the potential to run at 30 frames per second, and, to maximize game performance, that is the speed at which a competitive SPL team's software should operate. The faster the robot can interpret its surroundings and react, the better it will be at soccer. Even a frame rate of 20 frames per second can result in poorer performance as the robotic soccer player is slower to react to the game. This gives only 33 milliseconds (ms) to process each vision frame and to perform other tasks, such as motion calculations and communications. In practice, the vision processing segment of the program should last fewer than 20 ms per frame. These real-time requirements tightly constrain the software processing and spur innovation and optimizations in the vision systems.

1.0.2 Environment

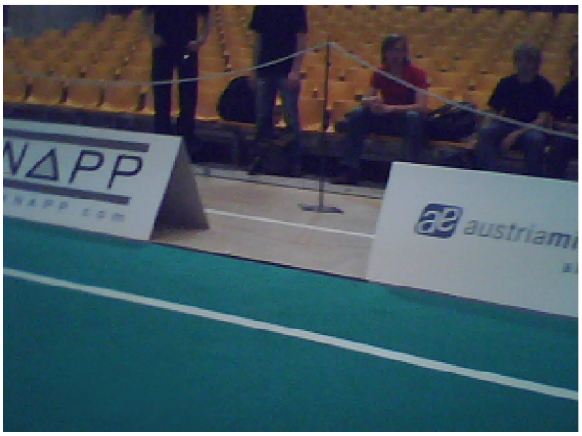


Figure 2: Confusing off-field objects

Not only do conditions within a game affect the performance of the vision system, but the changes between different competition settings, even within the same venue on the same day, can have a large impact on a vision system's performance. Lighting conditions in the venue can change when lights are dimmed or a cloud

obscures the sun through an open window. Many vision systems require long periods of human calibration in order to function correctly, even that of the competitive SPL team, the Bowdoin College Northern Bites. Their system, described in Section 1.1.1, requires ten or more man-hours of calibration for a new color lookup table at each competition. Different competition venues have different objects surrounding the field which can also confuse naive vision systems, so the vision processing must be intelligent about processing objects off the edge of the field. For example, in the vision frame seen in Figure 2, where the orange bleachers could resemble a large ball and the sign on the sideline could resemble a large line.

1.1 Related Work

1.1.1 Northern Bites' Vision System

The Bowdoin College Northern Bites' vision system is one of the more robust and reliable systems in the league, as evidenced by the team's top eight finish last year and top four finishes in previous years. It relies mainly on color segmentation and connected regions to form visual object detections in the image. The Northern Bites use a scaled down 320x240 image, provided by the camera, in order to maintain a high frame rate. [10]

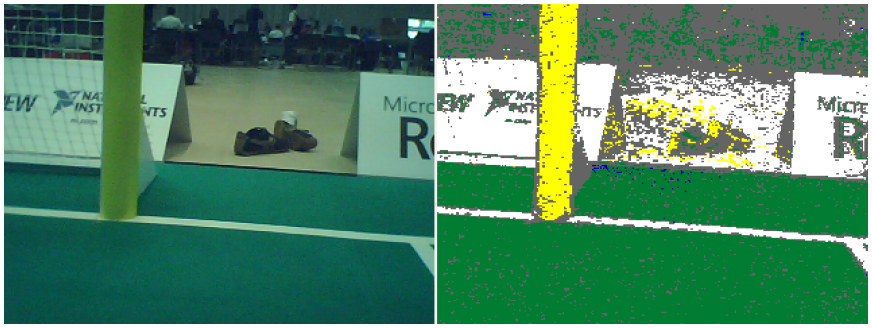
The Northern Bites' first step in vision processing is mapping each pixel in the image to one of the seven colors found in an SPL game: green for the field, yellow and blue for the goalposts, white for the field lines and robots, pink and navy for the uniforms, and orange for the ball. This mapping is generated beforehand using an offline tool and hand calibrating a lookup table for each venue and competition. The mapping produces a new image, based on the eight segmented colors, which is used for most of the rest of the image processing.

Once the color segmentation step has been completed, the next step in the vision pipeline is field line detection. For line detection, vertical and horizontal scans are used to find "line points." A line point is first identified by a rapid and large increase, or step edge, in the Y channel value of the original image. The line point is confirmed by checks looking for a transition in the color segmented image from green to white and back to green again. These line points are assembled and joined together using a variety of sanity checks, including the distance between line points, the angle with previously fitted points, and the absence of green between line points. Once each point has been tried in each candidate line group, some heuristics are applied to join lines which are likely part of the same line, add unused line points to the candidate lines, and extend lines past the final line point to fully fit the line on screen. All of this post-processing work is done using the color information.

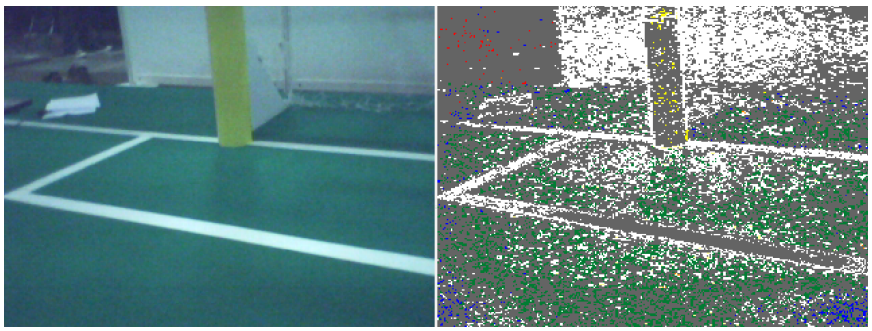
Field objects are found entirely using the color segmented image. First, vertical scan lines are performed to find long runs of colors which can represent goals, balls, or field crosses. Long runs of blue or yellow are used as likely locations for a goal post and "active scanning," examining pixels around the long run, is used to determine the shape of the post. The ball and field crosses are found using basic Run Length Encoding to collect runs of orange and white, respectively. The runs of color are

joined together to form blobs. The ball, goal post, and field cross blobs are each then passed through object specific sanity checks to ensure they are correctly identified.

The Northern Bites' current approach, like others in the SPL, is heavily reliant on color calibration. Color calibration is time intensive, error prone, and —in the Northern Bites system— a drag on performance. Slight differences in light brightness or color can have a profoundly negative effect on this color calibration. In Figure 3, the effects of changes in lighting are obvious. The color table created for RoboCup 2009 in Graz, where the image in Figure 3a was taken, works very poorly for the following image in Figure 3b, from RoboCup 2010 in Singapore. The illumination in each venue was different, resulting in a significant difference in the color of the field, goal posts, lines, and other important objects. Because of the lighting differences, each time the team arrives at a new venue, thousands of pictures are needed to ensure a properly calibrated color table.



(a) An image from Graz, color segmented with a color table built for it



(b) An image from Singapore, color segmented with the Graz color table

Figure 3: Two images from different venues, color segmented with the same table

The ad-hoc "sanity checks" are each composed of hand tuned parameters which are error prone, and case specific. The high number of parameters means that human programmers must spend long amounts of time tuning them to achieve proper results and any changes to the field setup affect the system in many places.

1.1.2 SPL Vision Systems of B-Human and rUNSWift

B-Human of the University of Bremen, Germany [4] and rUNSWift of the University of New South Wales, Australia [21] were the top two finishers in the 2010 SPL. Visual

abilities are the foundation of a good team, and these two teams succeeded because they had solid vision systems to use as bases for their behaviors and cognition. Both teams have released both their source code and detailed reports sharing the details of their vision system, as is common practice in the SPL.

Both B-Human and rUNSWift have similar approaches to their vision systems. Each team uses a region building algorithm to build regions based on color. They both downsize the image they use, in order to speed up processing. rUNSWift mainly uses a "saliency image" at a resolution of 160x120, a reduction of $\frac{1}{16}$ of the full size image [17]. In their region building, B-Human performs vertical scans on every eighth column with some special case scans in between for ball pixels. This greatly reduces the number of pixels scanned, but also reduces the accuracy of the scans. The regions constructed from these vertical scans are, in both systems, merged and filtered to construct accurate object detections.

These region building methods can effectively find objects on the field, but they suffer from many of the same problems of the Northern Bites' system. They are sensitive to color calibrations, meaning that each competition will require new calibrations to be made. Also, simply skipping columns, as B-Human does, wastes valuable information contained between the scan lines and can produce a very noisy representation of the image.

The region building algorithms employed by B-Human [20] and rUNSWift [17] use vertical scan lines to collect runs together into regions. Vertical scan lines are sensible in RoboCup, as objects like goals and robots are often vertically oriented, so more information is available scanning vertically, but vertical scan lines make very inefficient memory accesses. The image from the camera is represented as a large, row major 2D array of pixel values, and moving vertically through this array can cause many cache misses. Skipping vertical will skip forward in the array one row width's worth of pixels, possibly skipping past all the memory that had been cached and forcing the memory to load new pages from the disk. With the limited processing power of the Nao and the fixed processing time, these cache misses could degrade the speed significantly or force the resolution down, as is done with the sub-sampling.

1.1.3 rUNSWift Hough Transform Based Circle Detection

For the Open Challenge at RoboCup 2010, rUNSWift's entry was a vision system which could identify a black and white patterned mini-soccer ball. The system first classifies pixels as green, goal post colors (yellow and blue), or undefined. It then identifies the edges between green and undefined regions. Since the ball would be labeled undefined (it's black and white), a ring of edge points would surround it in the image.

By feeding it the green-undefined edge points, a Hough transform [13], explained in Section 3, was used to locate the ball in in the image. The Hough transform, when used for circles, is usually implemented with a 3D array, each dimension corresponding to the parameterization of a circle in an image:

1. x position of the circle center

2. y position of the circle center

3. Radius r of the circle

rUNSWift collapsed this 3D array to two dimensions, however, by using the knowledge of the ball's size and the kinematics of the robot. They computed the ball's expected size in the image at a given edge location, based on the distance to that pixel provided by the robot kinematics. Thus, the Hough transform could run in 40ms [17, 128]. This time is only for the circle detection and does not include any other object detection.

1.1.4 Nao-Team HTWK Open Challenge 2010

Nao-Team HTWK's entry for the Open Challenge at RoboCup 2010 was "Robust vision without calibration" [18]. The vision system they describe runs without any human calibration, so light conditions can change without too adversely affecting the system's performance. According to their report, they detect shapes instead of colors, and are able to run on a full 640x480 pixel image in less than 20ms. Unfortunately, their published materials on this topic are still forthcoming. We suspect, however, based on our personal conversations, that they are using techniques similar to those described in this document.

1.2 Overview

The current state of the art in RoboCup vision systems, mostly based on color segmentation and color run-length encoding, is ad-hoc and tied tightly to the RoboCup environment. The methods described by the top competitors in the league for identifying object shapes in the image are largely collections of hand coded heuristics for what defines a "goal post" or a "line" as a human would describe it. Domain information about the specific objects in question is, of course, required for reliable detection, but one goal of this project is to push the human intervention and "sanity checks" as far down the pipeline as possible, using general pattern recognition techniques first.

Computer vision research has produced many pattern recognition algorithms which are more reliable, more robust, and more general than these hand designed region building heuristics. General computer vision pattern algorithms are designed to work with the strengths of a computer, e.g. numerical accuracy and parallelism. Many general pattern finding algorithms, however, when naively implemented, are too slow to run in real-time on such a constrained platform as the Nao and there has not been sufficient research yet to optimize these algorithms for the settings. This is why teams in the SPL have been hesitant to use them in their software.

To put the performance requirements of a real-time vision system in context, consider that with a 640x480 YUV 422 image, there are 614,400 values to be processed. So, allocating 20 ms for vision processing, as described in Section 1.0.1, there is only 32 nanoseconds of processing time available per value. With a 500 MHz processor, like the Nao's Geode, that means there are only 16 machine cycles available per value. For comparision, a single integer divide can take 24 cycles, and one floating

point arctangent can take up to 354 cycles. [3] It is with these constraints in mind that this project was undertaken.

With a combination of algorithmic improvements and efficient programming practices, we believe that there are general algorithms which still hold promise in the SPL. SPL vision has often been solved in the past through the engineering of a solution specific to the needs at that moment. The "engineered" solution is often tied tightly to a certain property of the RoboCup environment, like the specific colors of the goals. This project seeks to bridge the gap between narrow solutions tailored to the situation and general algorithms with broader applicability by showing that a more general solution, like the Hough transform, can run in real-time on the Nao platform.



Figure 4: Image Processing Pipeline

The first step of the new vision processing is edge detection. Edge detection reduces the information in an image down to the outlines of objects. After the edge detection, the edge points are processed with the Hough transform. The Hough transform is a general purpose pattern finding algorithm, which will be used here to find lines. The lines found with the Hough will then be paired and trimmed to segments around the visible lines. High level heuristics will be used in the final stages to verify the lines. Once they are positively identified, field lines are important in robot self-localization.

This project aims to establish the Hough transform as a reliable and applicable vision algorithm for the embedded RoboCup Nao platform. The Hough transform has been used infrequently in work by other teams, and when it has, it was often in a limited or prohibitively slow manner. The Hough transform is an algorithm which suits the highly uncertain RoboCup environment well, and, when properly implemented, is also acceptable for full image processing on an embedded platform.

We seek a general algorithm for pragmatic reasons, not purely for theoretical improvements. General vision algorithms are robust in various circumstances and can be re-purposed and reused in new situations. Many current SPL vision systems tightly integrate the steps of object detection. When the rules change, or the environment is altered, the ad-hoc system can require extensive revising to be adapted. A good general algorithm will be easily adaptable. For instance, changing ball colors could require revisiting the heuristics used as a foundation of the current SPL vision systems. A general algorithm, on the other hand, should be easily adapted by simply changing a small portion of the system, like the metric used for gradient calculation. The current SPL vision solutions also skip lots of pixels, sampling the image down to a paltry 160x120 or below, in order to maintain high frame rates. General purpose algorithms maintain their accuracy by examining every pixel in the image.

In order to achieve the desired real-time efficiency of this project, we write the majority of the code for this project in x86 assembly language. Assembly language

code is often demonized as verbose, non-portable, and hard to maintain. These criticisms can be very true: about 150 lines of C++ edge detection code in this project was replaced with approximately 400 lines of hand written assembly language code which will only run in 32-bit mode on an x86 compatible processor, and has a steep learning curve. However, assembly language is often the only way to squeeze every bit of efficiency from a CPU. It lets the programmer use the ideal instruction mix for the task, a job a compiler can sometimes struggle to do.

Assembly language programming also forces a different programming paradigm on the programmer. Bill Silver, founder of Cognex Corporation and an experienced assembly language programmer, defends assembly language programming, saying,

> Programmers think in the language in which they program. Perhaps the central value of object oriented programming languages, for example, is not so much in the syntactic and semantic details of a given language but rather in way they encourage programmers to think about algorithms. So it is with assembly language, where programmers must think in the language of the machine itself. [22]

In addition, modern architectures have vector processing instructions which have no easily accessible and embeddable C/C++ equivalent. The vector processing units, known as SIMD units, are capable of operating simultaneously on multiple pieces of data, often with special instructions capable of performing complex operations with a single instruction. The Geode LX processor can operate the x86 MMX instruction set which uses 64-bit registers to operate on multiple 1, 2, or 4 byte values with a single instruction. We make extensive use of vector instructions throughout this project.

Assembly language code can be general and abstract, as well. The abstraction comes here not from the language but from the algorithms employed. Using well-defined interfaces between the steps of the line finding process means that the pieces can be reused later or easily modified to suit new situations. The performance gain is realized without losing generality of algorithm but rather by losing some generality of platform. Since the platform used is the x86 platform, however, the loss is slight. The x86 architecture is the most popular architecture in the world. Intel and AMD are committed in supporting backward compatability across the x86 product line, so all code written for the AMD Geode will run on any more modern x86 processor.

2 Edge Detection

Edge detection, as applied here, is the process of finding points of significant and rapid change in the intensity of one or more channels of an image. The physical boundaries between objects on the field cause brightness discontinuities which are captured in an image and are largely independent of lighting, viewpoint, and camera settings. This makes edge points a more reliable measure for object detection than the brightness values themselves.

Edge detection lowers the amount of information in an image significantly by reducing it to its most salient information. For example, a 320x240 YUV 422 image has 150 KB of information. [1] Reduced to a list of 9,000 edge points it will contain only approximately 52 KB of information. [2] By filtering out the noise and extraneous data from an image, the important information is more easily processed

There have been numerous algorithms developed for optimally finding the edges in an image. The more optimal, that is, accurate and noise-free, the edge detector is, the more computation it requires. [13] The Canny edge detector, an "optimal edge detector," for instance, requires multiple passes over an image at different resolutions to perform correctly. [9] In a computationally limited real-time environment, the extra computation outweighs the benefits.

To strike a balance between computational requirements and accuracy, we chose to use the Sobel operator [13] in the edge detection process. It requires only a single pass over the image, but performs very well as an edge detector.

As this project is concerned with detecting field lines, we chose to use the Y channel, or brightness, for edge detection, because the white field lines provide a stark contrast against the darker green of the field. The steep difference in brightness between the lines and the field is consistent between venues and in various lighting conditions. Even as lighting conditions change, the lines still provide a strong contrast against the field. This is ideal for RoboCup moving forward as we wish to become less tied to human calibration and better able to operate in uncertain and changing conditions.

The edge detection in this project is a four-step procedure. First, we estimate the x and y components of each pixel's gradient using the Sobel operator, a first difference operator that incorporates some smoothing to reduce noise. Next, we calculate the direction and squared magnitude of the gradient at each point. The squared magnitude is used to avoid a slow square root calculation. Then all points with magnitudes below a fixed noise threshold are discarded. Finally, we compare gradient magnitudes of the remaining points to find local maxima.

2.1 Sobel Operator

To detect the edges in images, we first calculate the gradient at every pixel in an image. The gradient is an approximation of the change across a pixel and has both a

[1] $(320 \times 240 \text{ pixels}) \times (2 \text{ bytes per pixel}) = 150\text{KB}$

[2] $(9000 \text{ points}) \times (6 \text{ bytes per edge}) = 52\text{KB}$

direction and a magnitude. To find the gradient we use the Sobel operator [13, 147] on the image's Y channel. The Sobel operator is a pair of kernels as depicted in Table 1, which approximate the (x, y) components of gradient.

$$\begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix} \quad \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ +1 & +2 & +1 \end{bmatrix}$$

(a) G_x (b) G_y

Table 1: x,y Sobel Kernels

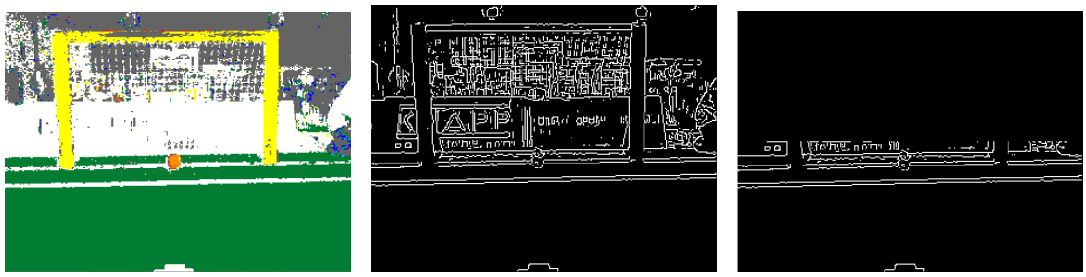
G_x and G_y, when applied around a pixel in the image, give the approximate gradient in the x and y directions, respectively. These kernels are applied by laying the kernel over the given pixel at the center of the 3x3 matrix, multiplying its neighbor pixels by their corresponding value in the kernel, and summing the products. This process is known as convolution.

The result of convolving the image is an estimate of a component of the gradient vector. The total magnitude of the gradient at each pixel is computed from these gradient values, as is the direction of the gradient. To compute the gradient map for the image, the kernels are applied to each pixel in the entire image. The x and y gradient components, along with the gradient squared magnitude are stored in a new array. The squared magnitude is used only for comparison, so its square root is unneeded. The mapping loses two rows and two columns from the image, however, because the kernel cannot be applied to the first or last rows, nor to the first or last columns. The 320x240 pixel image results in a 318x238 pixel gradient map.

To run the Sobel edge detector in real-time with the computational constraints of the Nao, the operation needs to be optimized significantly. The Sobel operator is an ideal candidate for SIMD instructions. G_x and G_y are calculated independently at each pixel. By hand coding in x86 assembly language and using the MMX instructions, we were able to calculate four pixels' gradients in parallel, resulting in a significant speed increase.

As processor speeds increase, memory accesses and memory bandwidth become a limiting factor in high performance computations. The 3DNow! instruction set provides the **prefetch** instruction to help lower the memory bottleneck. By bringing soon to be accessed memory into the fastest cache, the programmer can remove any waiting for I/O. By prefetching two rows below the current pixel during the Sobel operator execution, we reduced memory access penalties. When a new row is accessed, it is already in the L1 cache because the 64 KB L1 cache is large enough for more than four rows of pixels.

As the goal of edge detection is to reduce the image to the most salient points, we want to eliminate as much extraneous information as possible. In images where a large portion of the image is above the edge of the field, there were many extraneous edge points found in off-field objects or even in unimportant places like the net of the goal (See Figure 5b). To improve the output's relevance and decrease its size,



(a) The color segmented image (b) Edge detection on the entire image (c) Edge detection only performed below the field horizon

Figure 5: Reducing edges with the field horizon

and to improve the Sobel performance even further, the Sobel scan is limited to scanning below the highest section of the field in the image. In the image in Figure 5, by scanning only below the field horizon (Figure 5c), the number of edges detected drops by 73% and the Sobel operator is applied to only the relevant 49% of the image.

2.2 Edge Peak Detection

Once the gradient magnitude and direction is calculated at each pixel, the gradient values need to be compared to each other to find the actual edge points in the image. The edges are found by locating the peaks in the gradient map, the points of greatest change, which are the desired output of the edge detector. The goal of the peak detection phase is to also ensure that only one edge point is found for each real edge point in the image.

To find the peaks, the pixel gradient array is scanned. All pixels with a gradient magnitude below a fixed noise threshold are ignored so that only points of high change are considered.

For each pixel, the gradient direction is calculated as the arctangent of $\frac{G_y}{G_x}$ and stored. This information is not only useful in peak detection, but will also be used in line detection steps later.

Using the gradient direction, an asymmetric peak test is then applied in the gradient direction to all remaining candidate points. The peak test determines if the candidate edge point's gradient magnitude is greater than that of the pixel in the opposite direction of the candidate's gradient, and greater than or equal to the magnitude of the pixel in its gradient direction.

Figure 6 is an example, in one dimension, of the situation where the asymmetry of the peak test is important. The graph shows possible Y channel values and their gradient magnitude approximations. The maximum gradient magnitude, the true edge, falls here between position 3 and 4, and because of the discrete nature of digital images, positions 3 and 4 have equal gradient magnitudes. A purely symmetrical peak test would either fail to find any edges, if it used strictly greater-than tests, or it would find two peaks, if it used greater-than-or-equal tests. The asymmetric peak test, however, would find a single edge peak. Whether position 3 or 4 is labeled an

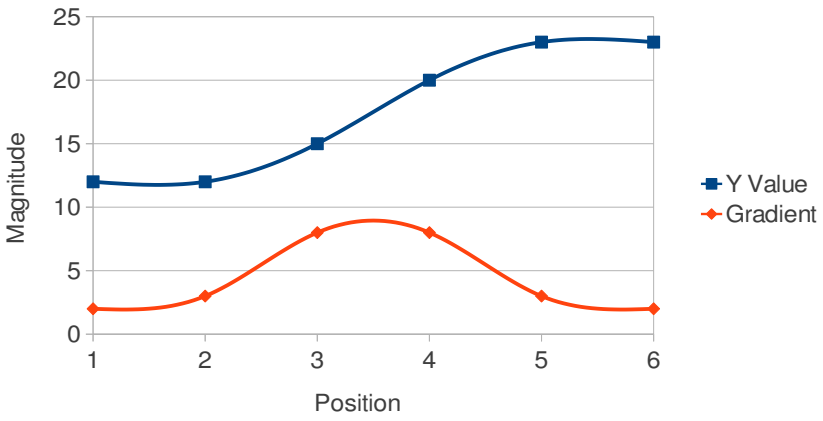


Figure 6: Asymmetric Peak Test

edge depends on the direction of the asymmetry, but, in practice, will not significantly affect the outcome.

Once a pixel passes the asymmetric peak test, it is labeled an edge peak and its location in the image, along with its gradient direction, are appended to the list of previously found edge peaks. The final list of edge peaks is then sent to the Hough transform to be processed further.

The peak test shrinks the image in by another two rows and columns. In order to test a pixel for peak candidacy, there must exist pixels on each side of the pixel being considered. Border pixels cannot be considered as peaks because they lack pixels on one or more edges. Thus, we begin the peak test with the second row and column of the gradient map and end it on the second to last row and column. The image is shrunk now to 316x226 pixels.

A significant limiting factor in peak detection computation time is the angle calculation. Floating-point calculations are slow and provide more accuracy than needed for our purposes. To remove this bottleneck, we chose to replace the floating point angles with eight-bit binary angles. An eight-bit binary number represents the range 0-255, so a radian angle t can be computed as an eight-bit binary angle with the formula $t_{int} = t_{radian} * \frac{128}{\pi}$. These binary angles are accurate to 1.41 degrees.

The machine level floating point arctangent function is also too slow, taking up to 354 machine cycles. [3] By precomputing an arctangent lookup table, we removed the costly arctangent function call. The table was indexed by the result of $\frac{G_y}{G_x}$. According to the Geode LX Databook, an integer divide of 2 byte-long values can take up to 16 machine cycles, while an integer multiply of similar length values only takes 3 cycles. [3] To take advantage of this, we precomputed another lookup table of $\frac{1}{G_x}$, indexed on G_x.

The power of using assembly language comes in the ability of the programmer to exploit every corner of a machine's instruction set. The x86 instruction set is large and has evolved over time to accommodate many different needs and applications. The advantage of the x86 instruction set's breadth is that it has many optimization

opportunities, but they can be hard for a compiler to use. The author of the code has the best idea what the code is intended to do, and the circumstances it will encounter, so he can choose the best instruction mix to accomplish that task.

There were three opportunities throughout this project to use relatively obscure x86 instructions to our great benefit. The first of these instructions, **repe scasw**, has two parts. The first part signifies that the instruction will repeat, without any external loop constructs, until the values it compares are equal. It automatically increments source pointer **edi** (by two bytes, according to the suffix "w," for "word"), decrements counter **ecx**, and compares register **eax** with the memory pointed to by the source pointer. This all happens faster than even a tight loop with separate loop control instructions, like Listing 2, can produce.

For instance, using:

```
  xor     eax, eax          ; Set eax to zero
  repe    scasw             ; Repeat scan while memory equals eax
                            ;   and counter is not 0
  jz      xLoop_end         ; If counter is zero, skip to the end
```

Listing 1: Looping with '**repe** scasw'

improves the entire edge peak detection runtime 16% over the equivalent piecewise code in Listing 2 which performs functionally the same operations, but which uses separate instructions that must each be decoded and executed in separate stages. The former loop does not advance the CPU pipeline until the '**repe**' condition is no longer satisfied. In this case, when the values are no longer equal or the counter in **ecx** is zero, the instruction exits.

```
  xor     eax, eax          ; Set eax to zero
start_loop:
  add     edi, 2            ; Move pointer forward 2 bytes
  dec     ecx               ; Decrement the counter
  jz      xLoop_end         ; Jump to the end if counter is zero

  cmp     eax, [edi-2]      ; Compare last value with zero
  je      start_loop        ; If the memory equals zero,
                            ;  repeat the loop
```

Listing 2: Separate instruction version of Listing 1

The **cdq** instruction is another example of a useful single instruction. The **cdq** instruction sign-extends the value in **eax** into **edx**, making **edx** all zeros or ones, depending on the sign of **eax**. Using this instruction, we can perform an absolute value in three instructions (See Listing 3), saving yet more time. On a two's complement machine, to negate a value, the machine inverts it and adds one. This method is also effective because it avoids a conditional branch which cannot be predicted by the hardware.

```
cdq                     ; if eax > 0 then
                        ;   edx is 0, else edx is all 1's
xor     eax, edx        ; Invert eax if it is negative
sub     eax, edx        ; If eax was negative, then subtract -1
```

Listing 3: Absolute value using **cdq**

Yet one more interesting instruction used to advantage in this project is **adc**. The **adc** instruction adds the carry bit from the last arithmetic operation onto the source register, and then adds that to the destination register. This operation is used to quickly tally up the high three bits of gradient direction needed to carry out the peak tests. It is possible to simulate this instruction in C/C++, but there is no way to rely on a compiler recognizing the operation as an add-carry. It suits our purposes very well, however, for quickly computing the three bit gradient quadrant.

3 Hough Transform

The Hough transform was patented in 1962 as a "method and means for recognizing complex patterns in photographs" [11], specifically straight lines, from a set of points. The transform has also since been generalized to find any parameterizable general pattern, [5] but we will only use it here for line finding.

A line in two dimensions can be uniquely defined in terms of $\{r, \theta\}$, where r and θ are the radius and angle, respectively, from the origin of the image to the closest point on the line. The Hough transform uses a 2D accumulator grid in the space $\{r, \theta\}$ as a voting mechanism for line finding in the image. [13] Each bin in the accumulator space represents a line in the image with components $\{r, \theta\}$.

The accumulator space holds the votes for every edge peak found in the edge detection step. It has dimension 256 in θ and 320 in r. For θ, 256 represents a full 8-bit binary angle, in the radians range (0, 2π]. r is a value in the range (-160, 160] chosen because it provides a good trade-off between speed and capturing all lines in the image, as it takes longer to operate on a larger accumulator space. In order to capture every possible line in an image, a maximum radius of 200 would be required, as that is the distance from the origin, the image center, to each corner of the image. However, we choose to ignore lines farther than 160 pixels from the origin because they are too short to be reliably detected.

Note that the Hough accumulator space is actually cylindrical. r has a fixed range which is discontinuous at the edges, but θ values 0 and 255 are actually adjacent in the accumulator space. Thus the Hough space actually wraps around on itself, a property which must be dealt during the Hough transform voting and peak detection.

Using full 2π radian angles and radii that can be positive or negative may seem redundant, but it gives the system the ability to distinguish between two parallel lines with a gradient in opposite directions. In Figure 7, the 2 lines are parallel, but as the gradients are opposing each other, they are represented by 2 distinct locations in the Hough accumulator, with angle values approximately 128 apart.

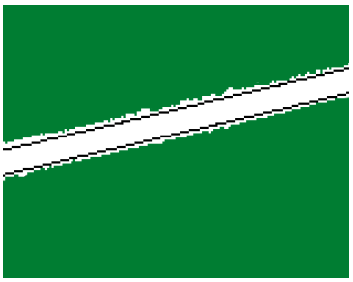


Figure 7: Parallel lines with opposite directions

The Hough transform is ideal for line finding in RoboCup. The field lines in RoboCup are uniform, straight lines but are often occluded. The Hough transform deals well with occlusion through the voting mechanism. [13] A line which is partially covered by a ball, or a robot, will still be recognized as crossing through the body of the robot. This is necessary for RoboCup vision as other robots are often obstructing

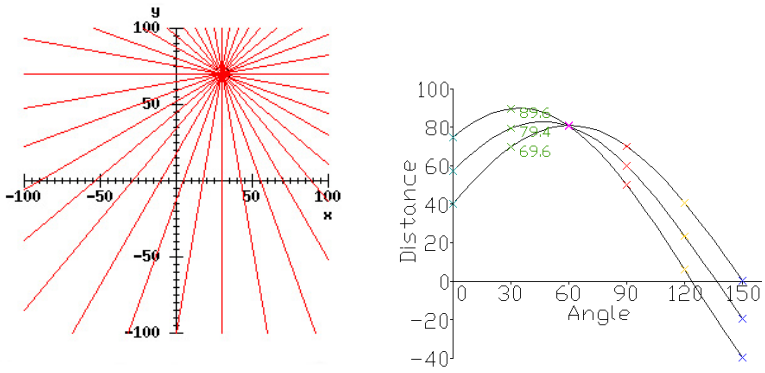
lines, as in Figure 11.

The generality of the Hough transform is a desirable element in a robotic vision system. It is decoupled from the edge finding routine that feeds it. The Hough transform requires only a list of points with x, y, and gradient direction which could come from brightness based edge detection, as they do here, or, for instance, color based goal post edge finding, as may be used in the future.

Without specific attention paid to its implementation, the Hough transform can be memory and processor intensive. [13, 17] which has limited its adoption in the league in the past. Specific attention was paid to its memory access patterns and computation ordering during the Hough voting procedure.

3.1 Voting

The first step in the Hough transform is marking the Hough accumulator space for every peak found during the edge detection. Looping through the list of edge peaks, we increment the $\{r, \theta\}$ accumulator bins which correspond to each possible line which could pass through that edge peak. Figure 8 shows how multiple lines can pass through a single point in an image at various angles and radii. Figure 8b shows the locations through the Hough space which are incremented for a given image point. Each line corresponds to a peak in the image.



(a) A single pixel could potentially lie on any of a number of lines [2]

(b) Incrementing the Hough Space for three points [1]

Figure 8: Example of lines passing through a point

A basic implementation of the transform would increment the bin for every line that could possibly pass through that line. For a single edge peak, over 256 bins would need to be incremented, at least one per angle. Figure 9 is a visualization (pixel intensity corresponds to bin value) of a Hough accumulator filled in such a manner. Filling so many bins would quickly overwhelm a real-time system. An image with 2,000 edge peaks would have to fill many more than 512,000 (and experimentally, likely more than 1,000,000) bins.

However, the gradient angle provides sufficient information to limit that search to within a small error range. The gradient direction approximation computed in

previous steps is approximately equal, within an error margin, to the angle of the line, so we can limit the marking in the Hough space to angles which are within a predetermined error margin of the gradient angle. The error margin needed is small, ± 5 was found sufficient for this project. Thus, instead of incrementing bins for 256 angles, only bins corresponding to 10 different angles, representing a span of approximately 14 degrees, are incremented.



Figure 9: Hough space marked with full angle

While limiting the span of Hough bins to mark is a significant improvement in running time, it is not significant enough, and there is still more time to be recovered. Thus far all the algorithms and implementations have operated sequentially in a single pipeline using only basic integer instructions. The Geode actually has two hardware pipelines, an integer unit (IU) and a floating point unit (FPU). The FPU handles MMX instructions, as well, reusing its 64-bit registers for SIMD commands. With its dual pipelines, the Geode is capable of executing an integer instruction and an MMX instruction every clock tick.

With this in mind, we redesigned the Hough voting procedure to intersperse MMX and integer instructions. By injecting a mix of instructions into the processor, we are able to interleave computations and significantly decrease the necessary run time of the routine.

We set up two pipelines which run simultaneously and communicate using a pair of **ping-pong buffers**:

1. **MMX/FP Unit:** The MMX registers compute the locations in memory to be incremented.

2. **Integer Unit:** The IU increments the Hough bins at the locations computed by the MMX pipeline.

The pipelines operate on the principle that the bins being incremented for a given peak are laid out in a predictable fashion. Using an error bound, `angleSpread`, and the gradient direction of the peak, t, we increment the bins over a span of $2 * \texttt{angleSpread}$ angles from $t - \texttt{angleSpread}$ to $t + \texttt{angleSpread}$. At each angle, we compute the radius r of a line passing through the point at that angle, and

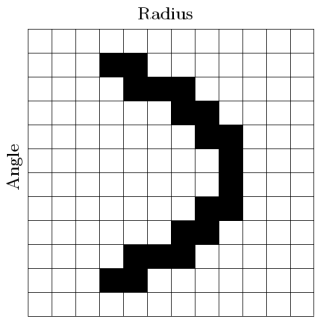


Figure 10: Example of one edge peak's votes in Hough space

increment the bins up to the radius at angle $t + 1$. The radius of a line at angle θ is $y * \sin\theta + x * \cos\theta$. Figure 10 shows a possible layout of the bins incremented for a single point. Incrementing across the r values creates a continuous line of incremented bins for each peak.

For a peak with gradient angle t, the MMX pipeline computes the radius at t and the radius at $t+1$. By using a lookup table with precomputed sine and cosine values and a unique MMX instruction, **pmaddwd**, we are able to quickly calculate the radii. Meanwhile, the IU pipeline loads in the location and span of the last computed row, and prefetches it. This effectively removes any stalls waiting for memory. Finally, after the loading is done and the MMX pipeline has finished computing the next span, the `RLoop` routine increments the bins starting with the radius at t and proceeding for the computed span length at t.

```
for(int i=0; i < angleSpread*2; i++){
  for(int j=0; j < houghWidth; j++){
    int sum = hough[i][j] + hough[i+256][j];
    hough[i][j] = hough[i+256][j] = sum;
  }
}
```

Listing 4: Wrap around the Hough space

The Hough space is laid out in memory in a two-dimensional array, but is actually a cylindrical space, where $t = 0$ and $t = 255$ are adjacent. Thus, when incrementing a point with a gradient direction within `angleSpread` of 0, it is important to handle the wrap around correctly. We solved this problem by extending our Hough space by `angleSpread` rows before $t = 0$ and after $t = 255$. This way no special wrapping code is needed to increment the space at a boundary point and no costly branching is needed. If $t -$ `angleSpread` is less than zero or $t +$ `angleSpread` is greater than zero, it simply writes into the buffer rows above and below the Hough space. The final step is to sum the values in the buffer rows with the values within the (0,255) space and write them back into memory, as in Listing 4. The copying and summation operations are performed quickly using MMX instructions.

$$\begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}$$

Table 2: 2x2 Boxcar Kernel

After the voting is complete, we smooth the Hough Space to reduce the noise. The smoothing is completed with a 2x2 "boxcar" kernel, as seen in Table 2. The kernel is applied to each bin in the accumulator by averaging each bin with the bins to the right, to the right and below, and below it. This helps to smooth noise in the Hough space due to errant edge peaks. Note that this shifts the Hough space right and down by a half bin, which is dealt with in later processing steps.

The boxcar is simple, fast, in place, and can be easily implemented using MMX instructions. Using a Hough space bin size of two bytes, four bin values can be loaded into a single MMX register. Listing 5 shows the additions which apply the boxcar operator. The minimum bin value threshold is then subtracted from the sums, limited to zero. These values are then written back into the original bin in memory.

```
; Load first row |00|01|02|03|04|
;       mm0=|00|01|02|03|
;       mm1=   |01|02|03|04|
movq  mm0, [esi]
movq  mm1, [esi+2]

; Add to 2nd row |10|11|12|13|14|
;      mm0= |00+10|01+11|02+12|03+13|
;      mm1= |01+11|02+12|03+13|04+14|
paddusw mm0, [esi + yPitch]
paddusw mm1, [esi + yPitch + 2]

; Sum to get final values
;      mm0= | 00+10+ | 01+11+ | 02+12+ | 03+13+ |
;           | 01+11  | 02+12  | 03+13  | 04+14  |
paddusw mm0, mm1

; Subtract threshold (mm7),
; Saturate to zero if below threshold
psubusw mm0, mm7

; Write out all four pixel sums to current bin
movq [esi], mm0
```

Listing 5: Smoothing with the boxcar kernel, **esi** is the address of the current bin

3.2 Peak Detection

After the Hough space has been filled, the bins with the highest values must be found. First the Hough space is scanned for bins with a non-zero value. During the smoothing step, we subtracted a fixed noise threshold and clamped the value at zero. Once a bin with a positive value is found, the eight bins surrounding it are compared with the current bin. If the current bin has a value greater than its eight surrounding bins, it is marked as a peak and added to the end of a list of peaks. Its r and t indices, along with its "score," the value of its accumulator bin, are recorded.

3.3 Line Candidate Narrowing

After finding all the local peaks in the Hough space, we need to remove any peaks which represent duplicate lines. For efficiency, the peaks are stored in an array, with their "activation" state held in a second array. On the first scan, all points within an angle of `angleSpread` are checked against each other. If any pair of these candidate lines are also within some small radius bound of each other, the line with a smaller score is deactivated, i.e. ignored in all future processing.

After duplicate lines are deactivated, the lines are searched once again looking for pairs. Note that a field line is defined by two step edges: one green to white, another white to green. Thus, each field line corresponds to a pair of lines in Hough space where the two lines have angles approximately π radians apart. The "Hough lines," as we will call them, are each compared to the others looking for the closest pairs of parallel Hough lines. We look for only the closest pairs of Hough lines because there could be multiple parallel field lines in an image, and it is desirable to only pair together the closest lines which actually outline a field line. All Hough lines which are not found to have an opposing parallel line are deactivated. This leaves only pairs of parallel Hough lines remaining. The list of line pairs is then returned for further processing.

The algorithms suppress duplicate lines and pair lines together are theoretically $O(n^2)$ in run time complexity, as they compare all lines with each other, but in practice, this is of little importance. After peak detection there are typically less than two dozen Hough lines remaining, but also, by only comparing lines which are within `angleSpread` angle measurements from each other, we cut the number of comparisons even more significantly. We know that the peaks in the Hough space are found in order of increasing angle, so the inner loop can exit as soon as a line with an angle difference of greater than `angleSpread` is found.

4 Field Line Detection

After the Hough transform has identified lines in the image, the vision system will trim the lines down to the line segments they represent in the image. To truncate the lines to segments accurately, the entire length of the line must be scanned. Because the line could be interrupted many times by a robot or a ball, an approach such as a binary search is not possible and the whole length of the line must be scanned. Figure 11 shows such an instance. In this case, the goal is to detect the field line as a single line, rather than as two separate segments. The blue overlay shows the detected line segments. In RoboCup, there is no situation or location on the field where an obstructed line could represent two separate line segments, so we always interpret lines like this as a single line.

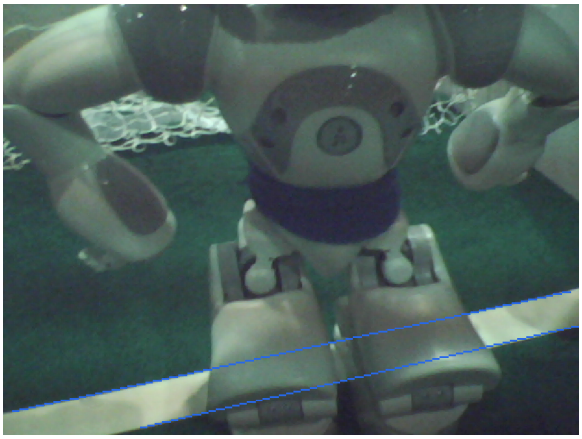


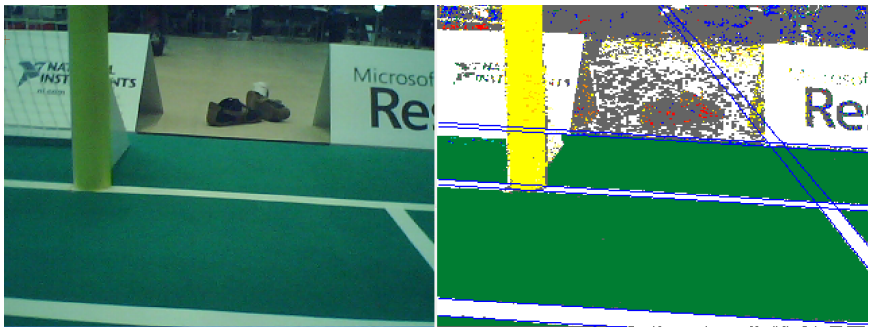
Figure 11: A line obstructed by a robot

The line segment end finding scan begins at one end of the line and scans the length of the line. At every pixel along the line, it scans perpendicular to the line for three pixels in each direction, looking for edge peaks with a gradient direction approximately equal to the line's angle in parametric coordinates. Each time a peak is found at a point in the line, it is counted as a hit, otherwise it is a miss. The scanner keeps a running count of the number of hits or misses in a row. When the count of hits first gets above a predetermined threshold, the location is marked as the beginning of the line segment. Once the segment beginning is found, the end is determined by finding the last point along the line where the miss count in the scan is less than a threshold. The threshold used to determine the line end is the same as the threshold used to determine its beginning.

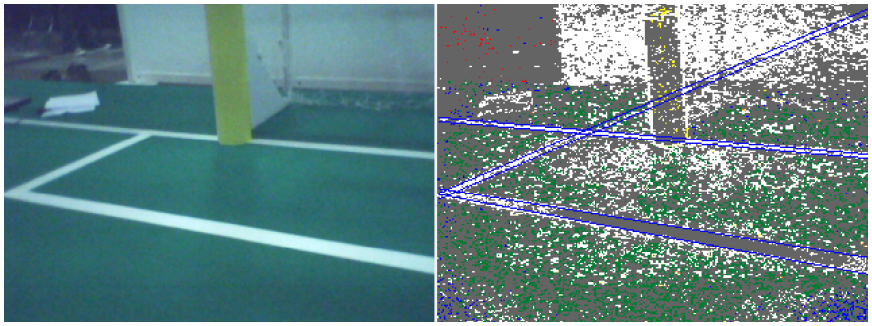
Once each Hough line is truncated to a line segment, field lines are defined by joining the pairs of lines, now line segments, created in Section 3.3. Having located the field lines in the image, the detection step is complete. The identified lines are used in other systems, such as field line corner detection, localization and robotic soccer behaviors.

5 Results

The Hough transform based line finder presented here is a step forward from the region and heuristic based approaches that are the current state of the art in the SPL. From edge detection through line pairing, this field line detection system is straight forward and effective. The entire system has only three parameters: the edge detection noise threshold, the Hough angle spread parameter, and the Hough noise threshold. Compared with the ad-hoc color based systems, which contain dozens of different parameters, the new system is more robust and does not require significant calibration.



(a) Hough line detection with a good color table



(b) Hough line detection with a poor color table

Figure 12: New line detection does not depend on color calibration

The new edge based system is also robust to changes in lighting and nearly color independent. In fact, color information is only used as a further optimization to detect field edges and limit the edge peak scan. The Northern Bites' current system, which relies heavily on color information, suffers with a poor calibration. The new edge detection and Hough transform based system requires only a small amount of field green calibration to perform at its fastest, though the same results can still be produced without any color information at all.

In Figure 12a and 12b, the advantage to a color independent system is obvious. These are the same images from Figure 3, with the results of the edge detection and Hough Transform steps overlaid. Even with a terrible color table, such as in 12b, the new line detection system described here is able to function properly.

The run times presented in this section were measured running on the Nao using

saved images from the Nao's camera. The images used were divided into four groups and can be seen in Appendix B. They range from devoid of lines and containing very few distinguishable edge peaks, to cluttered images which contain a significant number of edges. The run times were measured using the `clock_gettime` Linux system call, which provides nanosecond timing resolution. The clock tick counts were generated based on the AMD Geode LX in the Nao, which at 500 MHz ticks once every two nanoseconds. All C++ code here was compiled using GCC version 4.3.3 and the assembly was assembled using the GNU Assembler version 2.19.

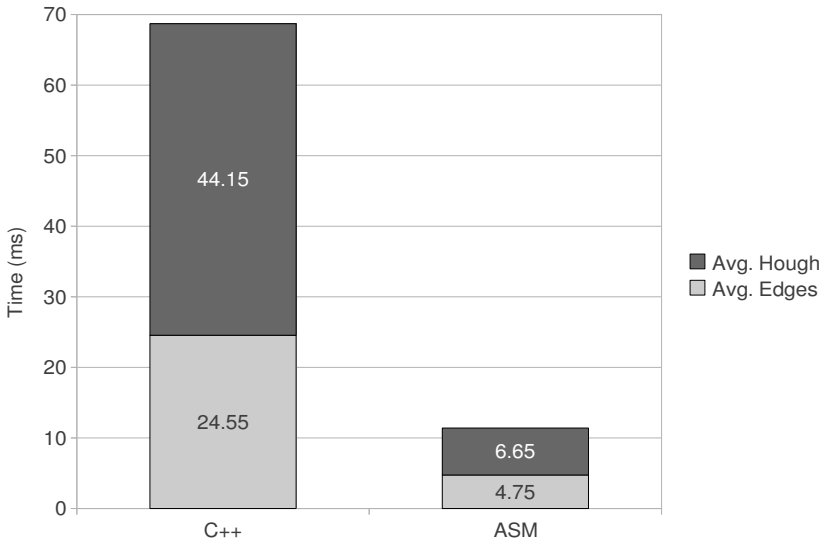


Figure 13: Average Full Image Line Detection Run Time

The new field line detection system using assembly language coded routines is fast. Compared with a C++ implementation of the same algorithm, it will run significantly faster while producing similar results. Differences between the two implementations come from floating point rounding and a small difference in gradient angle calculations between the implementations. When compared across a test set of 15 images with varying composition, the assembly version averages a factor of six improvement over the C++ version. Figure 13 compares the average run times over entire images (without the field horizon limiting mentioned in Section 2.1), breaking each version down into the edge detection and Hough transform steps. The C++ version averages a run time of 68.7 milliseconds per image, whereas the assembly version averages 11.4 milliseconds.

This project's goal is to build a Hough transform suitable for use in a robotic system running at 30 frames per second. At 30 frames per second, all the processing for each frame must occur in under 33 milliseconds, including not only vision, but also motion and behavior processing. A line finding system running at more than 15 milliseconds was decided to be unacceptable, as it left too little time for remaining processing. As such, the 11.4 millisecond average run time is acceptable for use in a real-time RoboCup SPL vision system.

Language	Image Set	# of Images	Avg. # of Edges	Edge Time	Hough Time	Total
C++	zero	3	57	21.2	7.0	28.2
C++	one	3	690	22.3	18.7	41
C++	couple	3	1362	23.4	31.6	55
C++	clutter	6	5949	31.3	119.3	150.6
ASM	zero	3	52	3.8	4.0	7.8
ASM	one	3	603	4.1	4.7	8.8
ASM	couple	3	1432	4.5	5.8	10.3
ASM	clutter	6	6127	6.6	12.1	18.7

Table 3: Average Image Run Times

Table 3 shows the test data for the full image run times. The tests were run 100 times on each image, and the times were averaged.

5.1 Edge Detection Performance

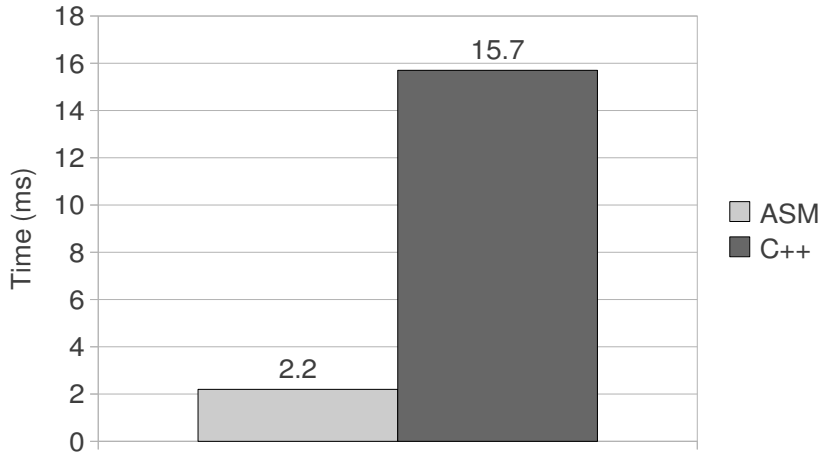


Figure 14: Average Sobel Operator Run Times

The edge detection step benefits the most from the transition to assembly language using the MMX instructions. A speedup factor of seven is achieved in the switch. As the Sobel operator is applied to every pixel in the image, the C++ and assembly raw timing values can be compared.

The combination of SIMD instructions and memory prefetching to improve cache hits yields a significant improvement. Memory access time is effectively cut to zero, leaving only computational time as a constraint. The SIMD instructions help to reduce even that by operating on multiple values simultaneously.

The peak detection step of edge detection is not a constant time for every image like the Sobel operator. Instead, its run time depends on the number of edge peaks found. Figure 15 shows the linearly increasing run times for the peak detection step

Image Set	Image #	Language	Avg. Time (ms)	# Edge Peaks	Clocks / edge peak	Clocks / pixel
zero	1	ASM	1.2	19	31578.95	7.81
zero	2	ASM	1.3	61	10655.74	8.46
zero	3	ASM	1.3	75	8666.67	8.46
one	1	ASM	1.6	508	444.52	10.42
one	3	ASM	1.7	646	331.41	11.07
one	2	ASM	1.6	654	325.27	10.42
couple	1	ASM	1.7	853	996.48	11.07
couple	3	ASM	1.8	1384	650.29	11.72
couple	2	ASM	2.2	2058	534.50	14.32
clutter	2	ASM	2.7	3037	444.52	17.58
clutter	4	ASM	3.9	5884	331.41	25.39
clutter	3	ASM	3.9	5995	325.27	25.39
clutter	1	ASM	4.2	6685	314.14	27.34
zero	1	C++	3.7	15	123333.33	24.09
zero	2	C++	3.5	76	23026.32	22.79
zero	3	C++	3.5	80	21875.00	22.79
one	1	C++	4.5	501	4491.02	29.30
one	2	C++	4.7	689	3410.74	30.60
couple	1	C++	5.1	723	3526.97	33.20
one	3	C++	5.2	881	2951.19	33.85
couple	3	C++	5.5	1425	1929.82	35.81
couple	2	C++	6.3	1938	1625.39	41.02
clutter	2	C++	8.6	3041	1414.01	55.99
clutter	4	C++	12.8	5663	1130.14	83.33
clutter	3	C++	12.5	5752	1086.58	81.38
clutter	1	C++	13.5	6629	1018.25	87.89

Table 4: Edge Peak Detection Run Times

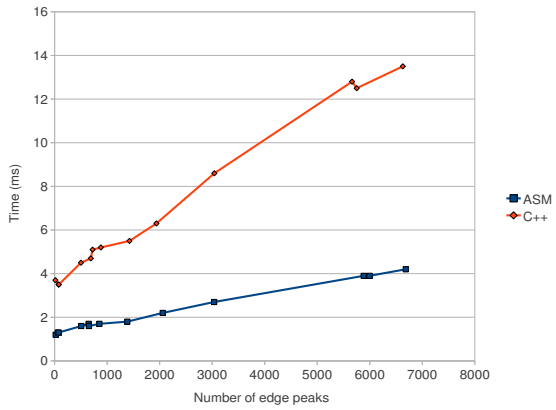


Figure 15: Average Edge Peak Detection Run Time

in C++ and assembly language. The assembly language version runs in less than one-third the time of the C++ version. Table 4 shows the run time data for the edge peak section.

5.2 Hough Transform Performance

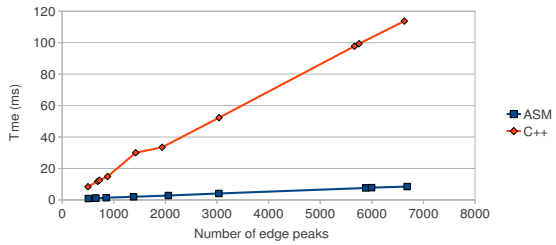


Figure 16: Average Hough Voting Run Time

The Hough voting routine, as measured in these experiments, is shown to be an acceptable real-time vision algorithm for RoboCup. The C++ version is indeed too slow for real-time use, as the busier images cause it to take over 100 milliseconds to complete. This is over three times the desired frame rate, so it is clearly not acceptable. The assembly version, however, takes under nine milliseconds, or one tenth of the time to complete on a similar number of edge points. This is a considerable speed up, given that there is no loss of accuracy between the two versions.

The Hough space smoothing routine is a simple operation involving four sums and a subtraction. The assembly version developed here runs over twice as fast as the C++ version. The output of GCC for this code, even using `-mmmx`, which allows the compiler to use MMX instructions, does not use MMX instructions and is thus limited to operating on one bin at a time. Efficient compiler generation of SIMD code is difficult [16] and programmers cannot rely on a compiler to generate any particular code if slight changes are made.

Image Set	Image #	Language	Time (ms)	# Edge Peaks	Clocks / edge peak
zero	1	ASM	0.3	19	7894.74
zero	2	ASM	0.4	61	3278.69
zero	3	ASM	0.4	75	2666.67
one	1	ASM	0.9	508	675.01
one	3	ASM	1.1	646	645.82
one	2	ASM	1.1	654	650.54
couple	1	ASM	1.4	853	820.63
couple	3	ASM	2	1384	722.54
couple	2	ASM	2.8	2058	680.27
clutter	2	ASM	4.1	3037	675.01
clutter	4	ASM	7.6	5884	645.82
clutter	3	ASM	7.8	5995	650.54
clutter	1	ASM	8.5	6685	635.75
zero	1	C++	0.3	15	10000.00
zero	2	C++	1.3	76	8552.63
zero	3	C++	1.4	80	8750.00
one	1	C++	8.4	501	8383.23
one	2	C++	11.8	689	8563.13
couple	1	C++	12.5	723	8644.54
one	3	C++	14.9	881	8456.30
couple	3	C++	30	1425	10526.32
couple	2	C++	33.5	1938	8642.93
clutter	2	C++	52.3	3041	8599.15
clutter	4	C++	97.7	5663	8626.17
clutter	3	C++	99.3	5752	8631.78
clutter	1	C++	113.7	6629	8575.95

Table 5: Hough Transform Voting Run Times

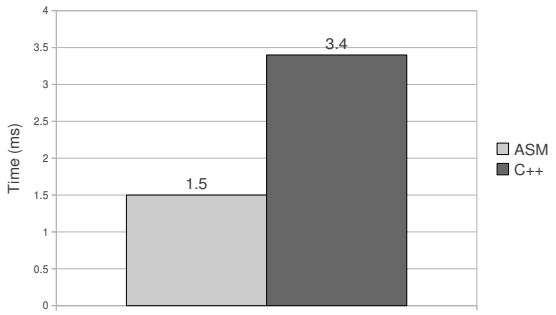


Figure 17: Average Hough Space Smoothing Run Time

The assembly language results presented thus far have been a significant improvement over the C++ versions, but they have still been slower than may be desired. Twelve milliseconds for line finding is longer than optimal. However, when the portion of the image scanned for edges is limited to the portion below the horizon, the speed up can be significant. As seen in Figure 5, the number of edges detected in some of the busiest images is cut by over 73%. The speed increase is generally more significant on images which contain more non-field objects. The field itself is typically not filled with many edges, so even if the field horizon is high in the image, it will generally contain few edge points, as seen in Appendix B, especially those in Tables 9 and 10. With the selective scanning enabled, this combination of edge detection and a Hough transform to detect field lines in RoboCup images is capable of real-time vision processing.

6 Future Work

In the future, the generality of edge detection and shape detection could be extended to the RoboCup goal posts and the ball. These objects are currently detected with color blobbing and present similar opportunities for improvements. A circular Hough transform would be possible with the ball, removing the dependency on color calibration to see the most important object in RoboCup vision. The orange ball presents a strong contrast to the green carpet in the V channel and presents a good opportunity to move slowly away from color calibrated information. The SPL plans to switch away from the orange ball to other styles of ball where the predetermined color is not important. A more general approach to ball detection would be helpful here.

This project shows the promise that general computer vision algorithms hold in RoboCup. They are adaptable to many situations and may not have the same calibration issues as the current approaches used in the SPL. We see that an efficient implementation can overcome barriers to the use of an algorithm and that SIMD instructions hold significant potential for embedded applications such as RoboCup.

For the Northern Bites, this project represents the beginning of a switch from engineered approaches to vision, which are built as custom-tailored solutions to the given scenario, to more general purpose solutions. Custom solutions have been chosen for their efficiency and the simplicity for the programmer, but they are inflexible by definition. They are designed for the RoboCup environment and are hard to extend into new domains. This project shows that general purpose solutions can run in real-time on the Nao and in other constrained environments.

The argument can be made that with faster processors there will be no need to implement solutions like this, but that argument falls apart when other changes are considered. For instance, doubling the dimensions of an image quadruples the pixel count. Higher resolution cameras can quickly overwhelm faster processors. A faster frame rate, higher accuracy demands, or even just more complex objects demand more processing efficiency and power than a faster processor can provide alone. Speed and necessity will always be an issue, so in order to move away from engineered solutions, efficient implementations of general purpose algorithms are necessary.

7 Conclusion

The transition to general image processing algorithms will improve vision processing robustness in changing environments, and is a necessary transition for the advancement of the sport of robot soccer. By replacing the color segmented and run-length encoding based system with a system using the Hough transform to identify field lines, we have taken a first step towards gaining independence from calibration and manual tuning. This will lead to increased productivity for the roboticist, as they do not have to spend so long calibrating the robot to new situations, and increased performance on the robot, as it can deal with fluctuations within its environment.

We have shown that general vision algorithms can still perform in this real-time environment and are a suitable replacement for the engineered solutions the SPL has typically used. Commonly held beliefs that the Hough transform is too slow for RoboCup are here shown to be baseless. A well designed system can be both robust and fast.

Also, computer scientists often shun hand written assembly as adding needless complexity and non-portability when optimizing compilers are available. As the results here show, an experienced programmer designing a complex system can still beat a compiler by making full use of the processor's architecture and instruction set in ways a compiler may not be able. The compiler's non-use of SIMD instructions is a great example. It is hard in a high level language such as C or C++ to instruct a compiler that it can make use of vector instructions for a certain complex task. By writing it oneself, a programmer can achieve optimal performance from the application.

Glossary

3DNow! AMD extensions to MMX.

MMX Intel SIMD instruction set which operates using 64-bit registers.

ping-pong buffers A pair of buffers are exchanged between two procedures operating simultaneously. While one procedure is writing to the first buffer, the other procedure reads from the second buffer. When they are both finished, they swap buffers and start again.

SIMD "Vector" instructions which simultaneously operate on multiple values packed into a single register.

SPL The Standard Platform League is the RoboCup league which competes only with the Aldebaran Nao.

YUV 3 channel color image encoding format with Y as luma and U, V as chroma components.

x86 Assembly Language Glossary

adc Sum the carry bit from the last operation with the src register then add onto the destination register.

cdq Convert Double to Quad. Sign extends **eax** into **edx**.

pmaddwd Packed Multiply Add Word. Performs two dot products on two vectors of length two: $|a|b|c|d| \cdot |e|f|g|h| = |a \cdot e + b \cdot f|c \cdot g + d \cdot h|$.

prefetch Prefetch at least a 32-byte cache line into the L1 cache.

repe scasw Scans through memory pointed to by **edi** until it finds a word (two byte) value that is not equal to the value in **eax**.

A Code

The C++ and x86 assembly language code created as part of this project is licensed under the LGPL as part of the Northern Bites RoboCup code repository and is available using the `git` versioning control system from `www.github.com/northern-bites`.

B Image Test Sets

The left image is the raw image and the right shows the results of Sobel edge detection.

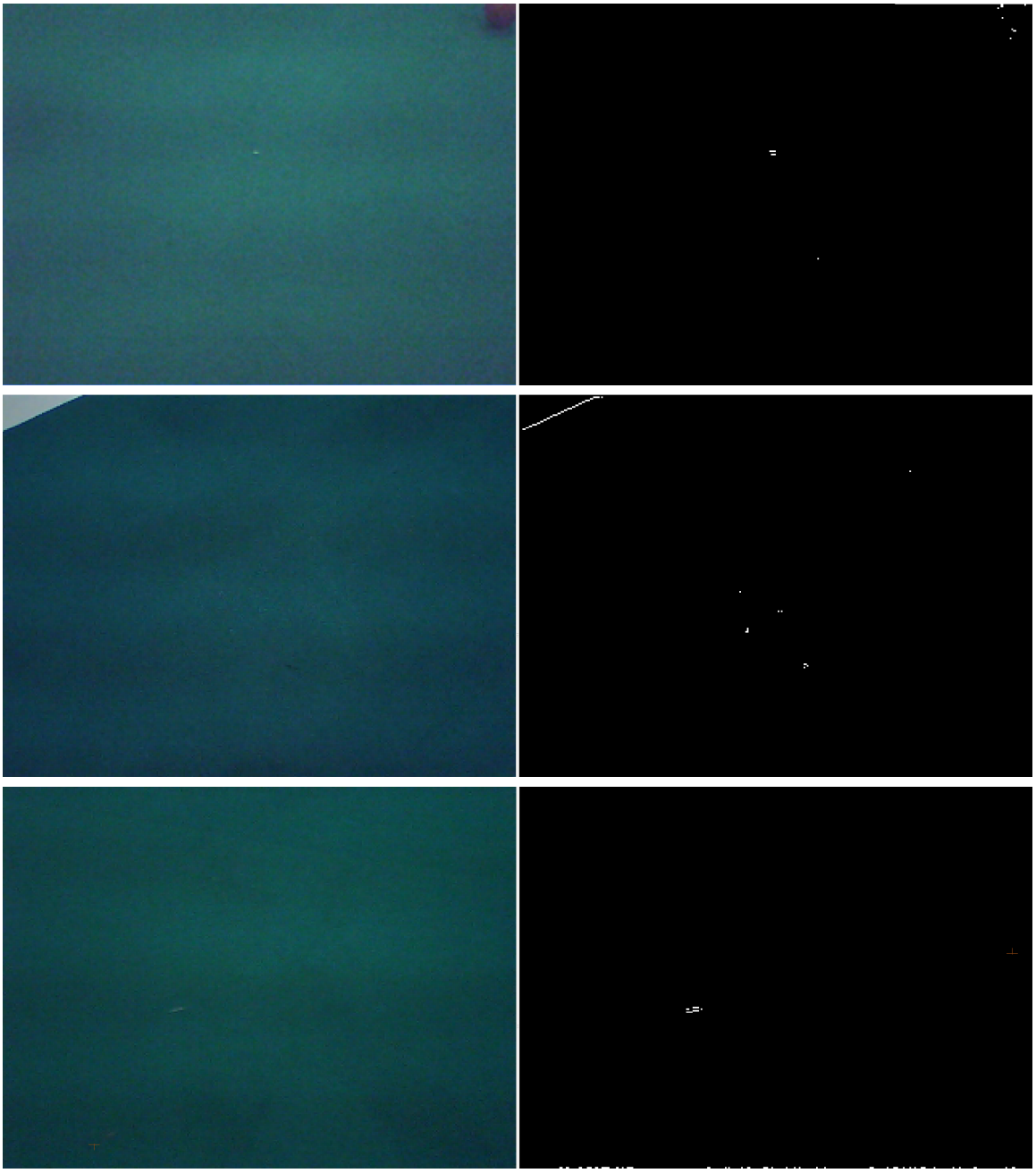


Table 6: Images with zero lines

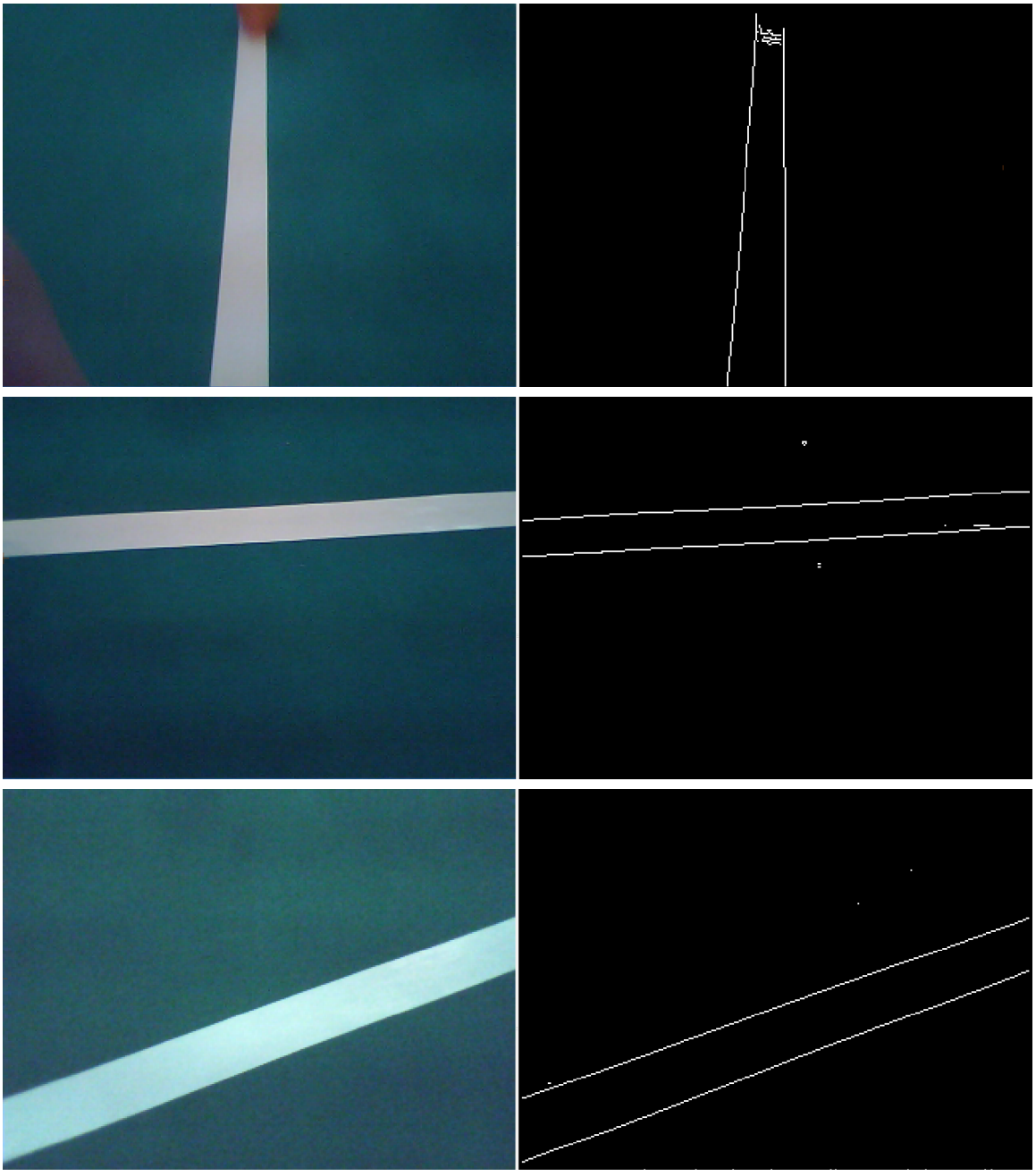


Table 7: Images with one line

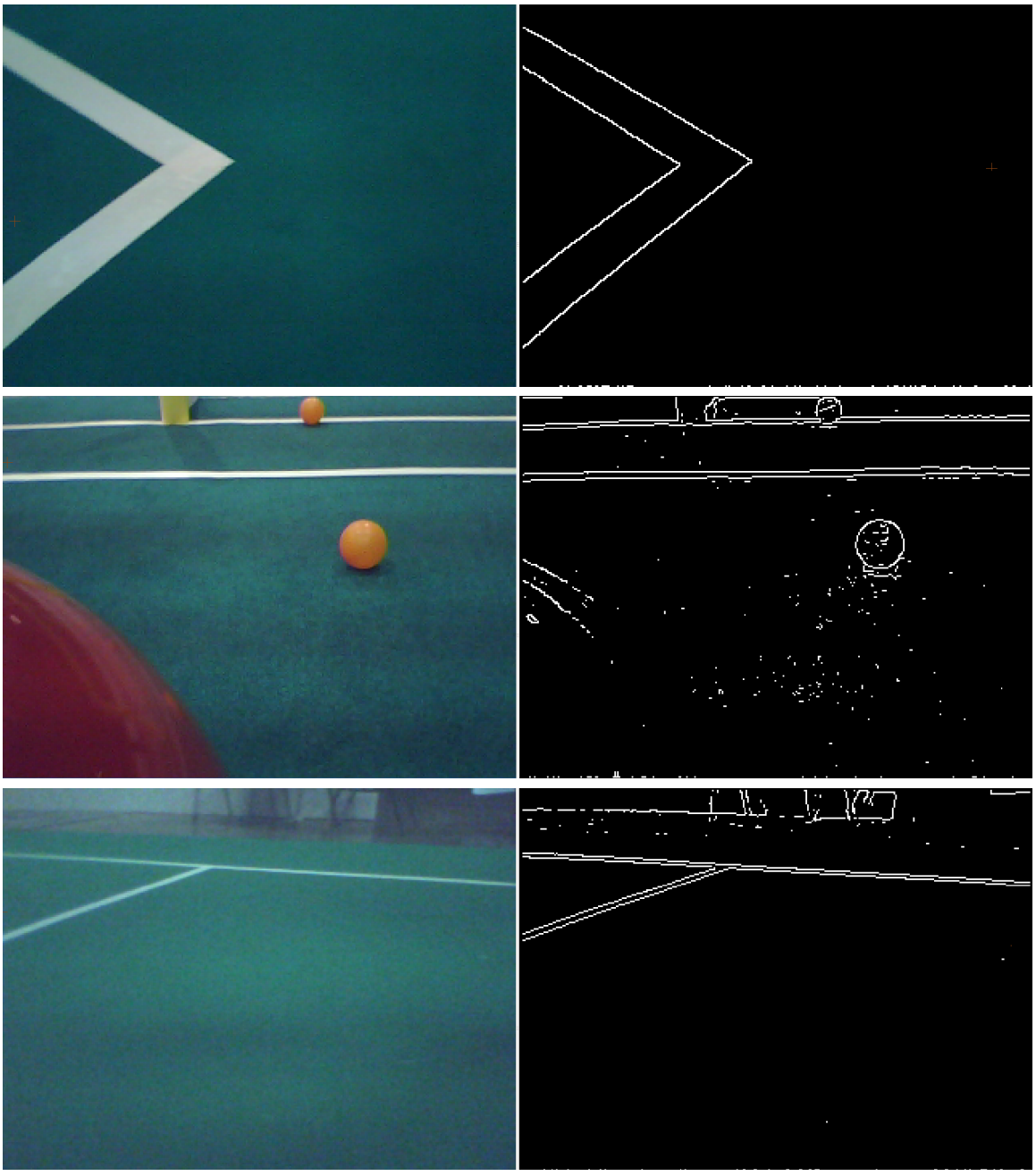


Table 8: Images with two lines

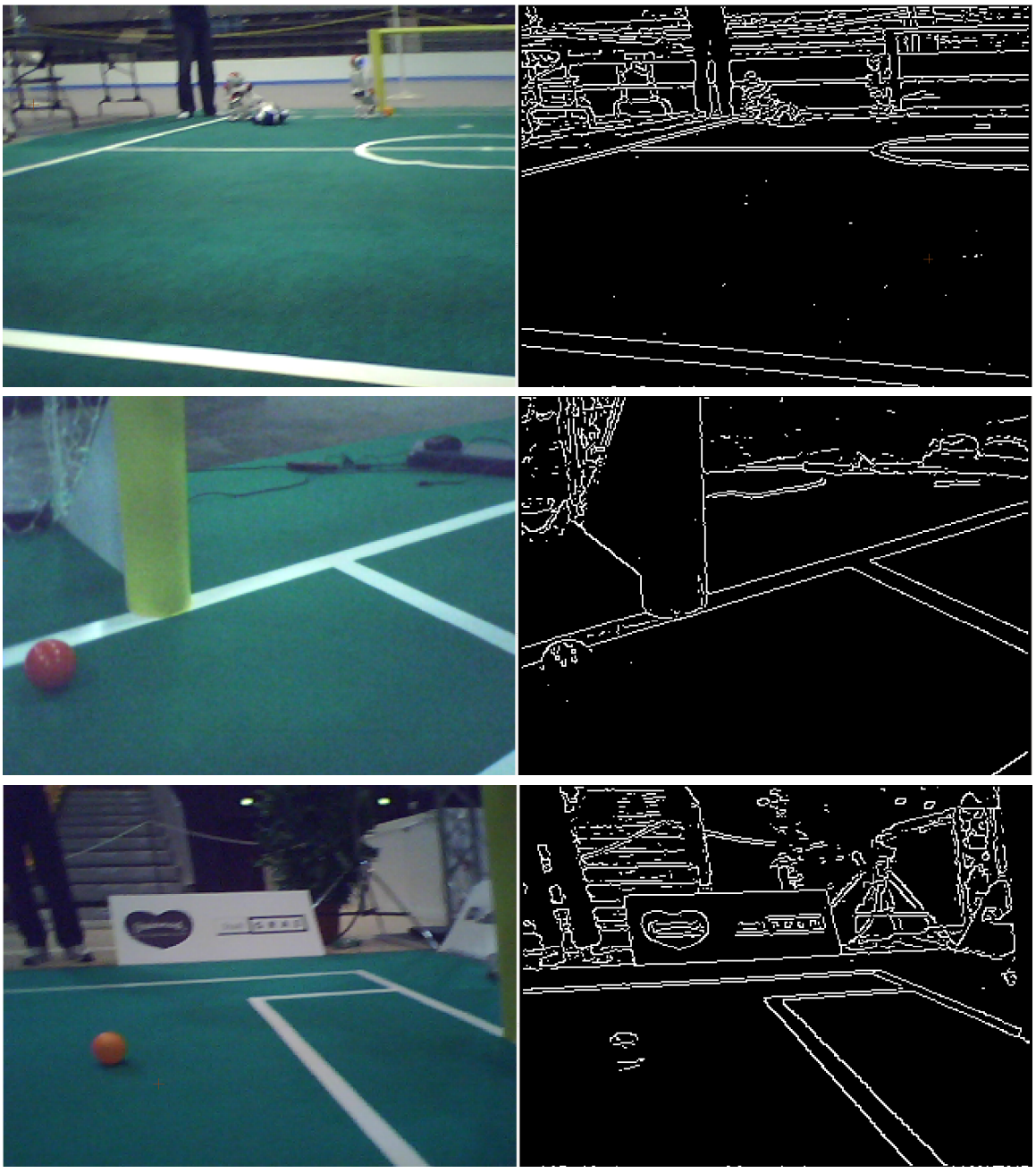


Table 9: Images with a cluttered composition



Table 10: More images with a cluttered composition

References

[1] http://en.wikipedia.org/wiki/Hough_transform.

[2] Peter e. hart, president and chairman, california research center. http://www.crc.ricoh.com/~hart/.

[3] Advanced Micro Devices, Inc., One AMD Place, Sunnyvale, CA 94088. *AMD Geode^{TM}LX Processors Data Book*.

[4] Team B-Human. http://www.b-human.de/.

[5] D. H Ballard. Generalizing the hough transform to detect arbitrary shapes*. *Pattern Recognition*, 13(2):111–122, 1981.

[6] Cognex Corporation. http://www.cognex.com.

[7] H Dahlkamp, A Kaehler, D Stavens, and S Thrun. Self-supervised monocular road detection in desert terrain. *Proc. of Robotics: ...*, 2006.

[8] G. N Desouza and A. C Kak. Vision for mobile robot navigation: a survey. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 24(2):237–267, February 2002.

[9] Martin A. Fischler and Oscar Firschein. *Readings in computer vision: issues, problems, principles, and paradigms*. Morgan Kaufmann, 1987.

[10] Tucker Hermans, Johannes Strom, George Slavov, Jack Morrison, Andrew Lawrence, Elise Krob, and Eric Chown. Northern Bites 2009 Team Report, 2009.

[11] Paul Hough. Method and Means for Recognizing Complex Patterns, 1962.

[12] Apple Inc. http://www.apple.com/ilife/iphoto.

[13] Ramesh Jain, Rangachar Kasturi, and Brian G. Schunck. *Machine Vision*. McGraw-Hill Science/Engineering/Math, 1995.

[14] Andreas Koschan and Mongi A. Abidi. *Digital color image processing*. John Wiley and Sons, 2008.

[15] RoboCup Standard Platform League. http://www.tzi.de/spl.

[16] Rainer Leupers. Code selection for media processors with SIMD instructions. In *Proceedings of the conference on Design, automation and test in Europe*, DATE '00, page 4–8, New York, NY, USA, 2000. ACM. ACM ID: 343679.

[17] Adrian Ratter, David Claridge, Bernhard Hengst, Brad Hall, Brock White, Benjamin Vance, Hung Nguyen, Jayen Ashar, Stuart Robinson, and Yanjin Zhu. rUNSWift Team Report 2010, 2010.

[18] Thomas Reinhardt and Stefan Seering. Nao-Team HTWK , Open Challenge Description : Robust vision without calibration, 2010.

[19] Aldebaran Robotics. http://www.aldebaran-robotics.com.

[20] Thomas Röfer, Judith Müller, and Tim Laue. B-Human Team Report and Code Release 2010, October 2010.

[21] Team rUNSWift. http://cgi.cse.unsw.edu.au/~robocup/.

[22] Bill Silver. personal communication, 2011.