Visual Inspection Of Railroad Tracks

2009

Pavel Babenko
University of Central Florida

Find similar works at: https://stars.library.ucf.edu/etd

University of Central Florida Libraries http://library.ucf.edu

Part of the Computer Sciences Commons, and the Engineering Commons

STARS Citation

https://stars.library.ucf.edu/etd/3990

This Doctoral Dissertation (Open Access) is brought to you for free and open access by STARS. It has been accepted for inclusion in Electronic Theses and Dissertations by an authorized administrator of STARS. For more information, please contact lee.dotson@ucf.edu.
VISUAL INSPECTION OF RAILROAD TRACKS

by

PAVEL BABENKO
M.S. University of Central Florida, 2006

A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the School of Electrical Engineering and Computer Science in the College of Engineering and Computer Science at the University of Central Florida Orlando, Florida

Fall Term
2009

Major Professor: Mubarak Shah
ABSTRACT

In this dissertation, we have developed computer vision methods for measurement of rail gauge, and reliable identification and localization of structural defects in railroad tracks. The rail gauge is the distance between the innermost sides of the two parallel steel rails. We have developed two methods for evaluation of rail gauge. These methods were designed for different hardware setups: the first method works with two pairs of unaligned video cameras while the second method works with depth maps generated by paired laser range scanners. We have also developed a method for detection of rail defects such as damaged or missed rail fasteners, tie clips, and bolts, based on correlation and MACH filters. Lastly, to make our algorithms perform in real-time, we have developed the GPU based library for parallel computation of the above algorithms.

Rail gauge is the most important measurement for track maintenance, because deviations in gauge indicate where potential defects may exist. We have developed a vision-based method for rail gauge estimation from a pair of industrial laser range scanners. In this approach, we start with building a 3D panorama of the rail out of a stack of input scans. After the panorama is built, we apply FIR circular filtering and Gaussian smoothing to the panorama buffer to suppress the noise component. In the next step we attempt to segment the rail heads in the panorama buffer. We employ the method which detects railroad crossings or forks in the panorama buffer. If they are not present, we find the rail edge using robust line fit. If they are present we use an alternative way: we predict the rail edge positions using Kalman filter. In the next step, common to both fork/crossings conditions, we find the adjusted positions of rail edges using additional
clustering in the vicinity of the edge. We approximate rail head surface by the third degree polynomial and then fit two plane surfaces to find the exact position of the rail edge. Lastly, using rail edge information, we calculate the rail gauge and smooth it with 1D Gaussian filter. We have also developed a vision-based method to estimate the rail gauge from a pair of unaligned high shutter speed calibrated cameras. In this approach, the first step is to accurately detect the rail in each of the two non-overlapping synchronous images from the two cameras installed on the data collection cart by building an edge map, and fitting lines into the edge map using the Hough transform, and detecting persistent edge lines using a history buffer. After railroad track parts are detected, we segment rails out to find rail edges and calculate the rail gauge.

We have demonstrated how to apply Computer Vision methods (the correlation filters and MACH filters in particular) to find different types of railroad elements with fixed or similar appearance, like railroad clips, bolts, and rail plates, in real-time. Template-based approaches for object detection (correlation filters) directly compare gray scale image data to a predefined model or template. The drawback of the correlation filters has always been that they are neither scale nor rotation invariant, thus many different filters are needed if either scale or rotation change. The application of many filters cannot be done in real-time. We have succeeded to overcome this difficulty by using the parallel computation technology which is widely available in the GPUs of most advanced graphics cards. We have developed a library, MinGPU, which facilitates the use of GPUs for Computer Vision, and have also developed a MinGPU-based library of several Computer Vision methods, which includes, among others, an implementation of correlation filters on the GPU. We have achieved a true positive rate of 0.98 for fastener
detection using implementation of MACH filters on GPU. Besides correlation filters, MinGPU include implementations of Lucas-Kanade Optical Flow, image homographies, edge detectors and discrete filters, image pyramids, morphology operations, and some graphics primitives. We have shown that MinGPU implementation of homographies speeds up execution time approximately 600 times versus C implementation and 8000 times versus Matlab implementation. MinGPU is built upon a reusable core and thus is an easily expandable library. With the help of MinGPU, we have succeeded to make our algorithms work in real-time.
This dissertation is dedicated to my caring and loving parents,

Iryna and Victor Babenko
ACKNOWLEDGMENTS

I would like to thank my advisor, Dr. Mubarak Shah, for continuous support and guidance over all of my graduate study years. I would not have finished this dissertation without his help and continuous encouragement. I would like to thank Dr. Ali Orooji, Dr. Takeo Kasparis and Dr. Xin Li for serving as my committee members and for their valuable comments and suggestions.

The majority of work performed in this dissertation was done in the scope of the FDOT projects – projects performed for Florida Department of Transportation by Computer Vision Laboratory at UCF. Experiments were performed on the railroad tracks of Central Florida with the help and permission of FDOT staff. I would like to specially note Don Harper whose hard work on hardware required for our experiments made this work possible.
# TABLE OF CONTENTS

LIST OF FIGURES .................................................................................................................. xi

LIST OF TABLES .................................................................................................................... xv

LIST OF CODE LISTINGS ..................................................................................................... xv

LIST OF SCHEMES ................................................................................................................ xv

CHAPTER 1  INTRODUCTION ................................................................................................... 1

1.1 Overview .......................................................................................................................... 1

1.2 The research approach and organization of the thesis .................................................... 5

1.3 The hardware used in experiments .................................................................................. 7

1.4 Summary ......................................................................................................................... 11

CHAPTER 2  LITERATURE REVIEW ...................................................................................... 12

2.1 The optical methods ........................................................................................................ 12

2.2 Laser triangulation methods ............................................................................................ 13

2.3 Visual methods ................................................................................................................ 14

2.4 Summary ........................................................................................................................ 16

CHAPTER 3  DETERMINING RAIL GAUGE USING PURE LASER APPROACH ............. 18

3.1 Introduction ...................................................................................................................... 18

3.2 Algorithm details .............................................................................................................. 20

3.2.1 Building 3D panorama of the rail ............................................................................. 20

3.2.2 FIR filtering and Gaussian smoothing of input data ................................................ 20

3.2.3 Segmenting out the rail surface by using k-means clustering .................................. 22
3.2.4 Verification of presence of railroad crossing or fork.............................................23
3.2.5 Finding the rail edge if forks/crossings are not present (Robust Line fitting) ........27
3.2.6 Finding the rail edge if forks/crossings are present (using Kalman filter prediction scheme) ..................................................................................................................27
3.2.7 Find adjusted positions of rail edges using mean shift clustering ......................29
3.2.8 Approximate rail head surface by the third degree polynomial .............................30
3.3 The method of evaluation ........................................................................................30
3.4 Summary ......................................................................................................................32

CHAPTER 4  DETERMINING RAIL GAUGE USING PURE CAMERA APPROACH ......33
4.1 Introduction ................................................................................................................33
4.2 Detecting the rail in the image .....................................................................................34
4.3 Estimating the distance between rails .........................................................................38
4.4 Results ........................................................................................................................39
  4.4.1 The evaluation data set..........................................................................................41
  4.4.2 Rail detection experiments ....................................................................................43
  4.4.3 Rail gauge measurements .....................................................................................45
4.5 Method for camera calibration ..................................................................................46
4.6 Summary ......................................................................................................................47

CHAPTER 5  OTHER VISION METHODS FOR RAILROAD DEFECT DETECTION ....48
5.1 Detecting Rail Fasteners .............................................................................................48
5.2 Locating Rail Fasteners .............................................................................................52
5.3 Detecting Missing Fasteners ......................................................................................54
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.4 Experiments</td>
<td>56</td>
</tr>
<tr>
<td>5.5 Detecting Rotated or Untypical Fasteners</td>
<td>59</td>
</tr>
<tr>
<td>5.6 Summary</td>
<td>61</td>
</tr>
<tr>
<td>CHAPTER 6 REAL-TIME IMPLEMENTATION</td>
<td>62</td>
</tr>
<tr>
<td>6.1 Introduction. MinGPU library</td>
<td>62</td>
</tr>
<tr>
<td>6.2 Current trends in GPU development</td>
<td>64</td>
</tr>
<tr>
<td>6.2.1 Graphics processors</td>
<td>64</td>
</tr>
<tr>
<td>6.2.2 GPU limitations</td>
<td>67</td>
</tr>
<tr>
<td>6.3 MinGPU library</td>
<td>70</td>
</tr>
<tr>
<td>6.3.1 ‘Hello, World!’ example</td>
<td>71</td>
</tr>
<tr>
<td>6.3.2 MinGPU operating modes</td>
<td>73</td>
</tr>
<tr>
<td>6.3.3 MinGPU Basic examples</td>
<td>74</td>
</tr>
<tr>
<td>6.4 MinGPU applied to railroad defect detection</td>
<td>79</td>
</tr>
<tr>
<td>6.4.1 MinGPU implementation</td>
<td>81</td>
</tr>
<tr>
<td>6.4.2 Time considerations</td>
<td>84</td>
</tr>
<tr>
<td>6.5 Discussion</td>
<td>85</td>
</tr>
<tr>
<td>CHAPTER 7 SUMMARY AND FUTURE WORK</td>
<td>90</td>
</tr>
<tr>
<td>7.1 Summary of Contributions</td>
<td>90</td>
</tr>
<tr>
<td>7.2 Future Work</td>
<td>92</td>
</tr>
<tr>
<td>LIST OF REFERENCES</td>
<td>94</td>
</tr>
</tbody>
</table>
LIST OF FIGURES

Figure 1: One of the rail cart designs we used in our experiments..........................8

Figure 2: The schematics of the cart we used in the experiment. The two lasers are installed over
each rail perpendicularly to the rails. The fields of view are known to move to further than +4/-9
degrees in every direction, so the camera views are reasonably constrained..........................9

Figure 3: Laser triangulation method in its essence. a) The laser projects a line onto the surface
of the rail, which, due to the rail geometry, is split into several lines on fields of view of camera.
b) The projected laser line after the processing stage. The edges of the line define the edges of
the rail........................................................................................................................................9

Figure 4: The profile of a single range scan acquired by the device in raw unaltered data. In blue,
the supposed position of the rail is overlapped on the image. In red we highlight points in the
vicinity of the rail. The device possesses intrinsic error in distance measurements of about +/-0.5
inch................................................................................................................................................10

Figure 5: Rail head close-up in the input data.................................................................10

Figure 6: The separation of the rail head using the 3-means clustering scheme. Due to the
geometry of the experiment the points roughly group around several clusters, namely rail head
and two side rail plates..................................................................................................................23

Figure 7: Railroad crossing in raw input data. The small groove is visible to the right of the
railroad rail. The railroad truck wheels fit into the groove, so it must be always present. This
groove is hard to detect by standard methods, so it needs special processing by Kalman filter
prediction mechanism..................................................................................................................24
Figure 8: The profile of a rail gauge measured in inches at a track segment of about 8000 feet long. In the experiment, we ran the cart over the same track more than once. We overlap the resulting gauge curves on this figure. We observe that overlapped gauge curves closely match. 31

Figure 9: The profile of the rail. The topmost part the wheels lay on is the rail head, while the bottom part attached to the ground is the rail base. According to US standards, the rail base is always about twice wider than the rail head. .............................................................35

Figure 10: a) sample input image b) vertical Sobel edges are generated from input image. .......35

Figure 11: A Hough transform of a Sobel edge map. All lines in Hough space which are co-parallel with the longest line in Hough space are located in the same column. We allow for some margin by including a few adjacent columns. We eliminate all other lines. Red rectangle in picture shows the (approximate) position of the cut-off in Hough space. .................................36

Figure 12: The tie plate which appears in this image to the right side of the rail generates strong edge close to parallel to the rail edge. .................................................................37

Figure 13: a) The candidate rail lines (green) are determined on step 2. b) The lines corresponding to rail head edges (yellow) are located in step 3........................................37

Figure 14: This figure shows an occurrence of different measured rail widths in a sequence of 200 images. X axis plots rail width, while y axis plots an occurrence. About 10 images produce significantly deviated rail width (represented by the second small spike on a figure). Those images are invalidated. .................................................................38

Figure 15: This schematic shows how we obtain rail gauge $G$ from estimates of rail positions in images. Arbitrary zero points on $x$ axis are chosen for each camera. The world distance between the zero points and camera zoom levels $Zoom_L$ and $Zoom_R$ are measured during camera
calibration. Zoom levels define a pixel – to – inch ratio for objects in camera view at a rail
distance. Rail edge offsets from zero points $Off_L$, $Off_R$ are measured for every image. Thus, the
rail gauge in world coordinates is found as $G = D_{cam} + Off_L \times Zoom_L + Off_R \times Zoom_R$.

Figure 16: Measured rail offset is an offset of an inner rail edge from the (right) camera zero
point..............................................................................................................................................39

Figure 17: A mosaic of images from each of 10 collections in our dataset. .........................42

Figure 18: The profile of a rail gauge measured in inches at a track segment of about 100 feet
long..................................................................................................................................................46

Figure 19: a) The calibration bar as seen from the left camera. The widths of black stripes and
the distances between them are known values of 1” and 2” respectively. The distance between
the rightmost stripe edge and the corresponding leftmost stripe edge in the right camera is also
known. b) The edges generated by the calibration bar c) The calibration bar detection. ............47

Figure 20: Clockwise from the upper left: two steel fasteners, a fast clip, an e-clip on a wooden
sleeper, and an e-clip on a concrete sleeper..................................................................................49

Figure 21: Confusion matrix for the fastener classes using OT-MACH filters.........................51

Figure 22: Detected fasteners of different types: a) steel fastener; b) E-clip (concrete); c) E-clip
(wooden). The overlaying numbers give a confidence score.........................................................52

Figure 23: a) Here a missing fastener is detected by direct correlation; b) Here a missing fastener
is not detected by direct means. ..................................................................................................55

Figure 24: This is a distance-between-fastener plot. Each integer index on the x-axis represents a
detected fastener. The y-axis is the distance between a fastener and the previously detected
fastener. The fastener that was not detected in Figure 23 can still be found by looking for deviations in the gaps between detected fasteners. ...............................................................55

Figure 25: Probability of detecting a fastener vs. probability of false alarm. The locations of the 70%, 60%, and 50% thresholds are shown.................................................................58

Figure 26: a) Left: An example of a "gray" steel fastener. Contrast it with the image to the right. Right: An example of a fastener whose confidence is reduced due to clipping. b) Left: An example of a missing fastener - a defect. Right: An example of an over-saturated image. ........58

Figure 27: This figure illustrates a possible application for GPU scaled correlation filters. UAV IR night video has occasional inclusions of people of unknown scale which must be detected. The video is pre-processed with thresholds and Harris corner detector, which leaves about 30-150 possible people candidates in every frame. The exact scale of people is unknown because aircraft altitude keeps changing; however, there exists a limiting range for the scales, both upper and lower. People are being detected by applying a set of differently sized correlation filters to every possible position. Resulting detections are then tracked to ensure their consistency. ........80

Figure 28: The correlation filters implemented on GPU. Railroad track video is made with a camera fixed over tracks, so the scale of objects in a field of view does not change. Our purpose is to detect and count rectangular rail fastener clips which hold the rail in place. However, besides being spatially rotated for up to 30 degrees, clips can be of slightly different geometric/visual appearances by manufacturer’s design. In a set of filters we put a filter for every possible filter appearance. Filters are created using OT-MACH algorithm with a rotation step of 3 degrees, and then applied sequentially to an image. .................................................................80

Figure 29: Time versus image size dependence for algorithms run on the GPU. .....................87
LIST OF TABLES

Table 1: Our dataset we used in experimental validation. ..................................................43
Table 2: Results of rail detection evaluation on data collection 6. ...........................................45
Table 3: The trends in GPU evolution in recent years. ..............................................................66
Table 4: Comparative execution times for homography transformations algorithm. ...............86

LIST OF CODE LISTINGS

Listing 1: introduction to MinGPU ..................................................................................71
Listing 2: reduced ‘Hello, World’ .......................................................................................74
Listing 3: image derivative in x, y direction........................................................................75
Listing 4: image derivative in t direction .............................................................................75
Listing 5: image derivative, unfolded loops ..........................................................................77
Listing 6: normalized cross-correlation ..............................................................................82
Listing 7: main formula in normalized cross-correlation ....................................................83

LIST OF SCHEMES

Scheme 1: The solution diagram to railroad rail edge detection and gauge measurement. .......19
CHAPTER 1

INTRODUCTION

1.1 Overview

The federal railroad administration provides data on railroad accidents throughout the USA on their website [FRA]. According to this database, between 2005 and 2008, 7582 train accidents occurred in the USA. Of these accidents, 1391 (18%) occurred due to the track geometry, 1251 (16%) due to the defects in track components, such as rail, fasteners, and joints, and 670 (9%) due to defects in switches. Faulty tracks account for more than one-third of all railroad accidents. Maintenance of the expanding network of rail tracks requires a sizable investment of time and effort. According to the Federal Railroad Administration’s Office of Safety Analysis, from 1995 to 2003, 45,842 inspections were performed nation-wide and 216,100 defects were recorded. Nonetheless, despite the high level of inspection effort involved, the continuing high accident rate raises questions about the extent to which the railroads comply with the inspection requirements as well as about the extent to which inspections can help to avert accidents. Great levels of performance can be achieved through the automation of inspection using computer vision systems, as they allow scalable, quick, and cost-effective solutions to tasks otherwise unsuited to humans.

A railroad is built from several basic components. Two steel rails are mounted in parallel, which supports the efficient motion of the rail cars. These rails are laid upon crossties, either wooden or concrete, which are embedded in ballast and laid perpendicular to the rails. Crossties span a
distance greater than the width between the rails. The tracks are supported by a flexible bed of ballast which might be a mixture of gravel and/or aggregate. Finally, the rails are anchored by fasteners or clips to the concrete crossties, and by tie plates to wooden crossties. The fasteners and plates can take many forms [AFM00]. Steel spikes hold tie plates in place in wooden crossties, while steel bolts are used to hold fasteners in concrete crossties.

Rails typically have a length of about 60 feet. Rails joints are where the two rails meet. Rails can be welded or unwelded at rail joints. Most railways in the USA built after 1950 use continuously welded rails. In these tracks, rails are welded together by using flash butt welding to form a single rail which can be thousands feet long. If rails are unwelded, they are held together by joints, bars, and bolts.

Basic track geometry parameters include gauge (roughly speaking, the distance between the rails) and cross-level (the difference in height of the rails). Curvature (the difference in heading of two locations 100 feet apart expressed in degrees) is another important track geometry parameter. Two additional parameters are alignment and profile.

Railroad tracks are laid out according to a mathematical model. Three types, interconnected to define a desired path, comprise the track. These types are referred to as tangent, spiral, and curve. Tangent segments are straight or linear, curved segments have a fixed radius, and spiral segments interconnect tangent and curve sections.

The term “track defect” encompasses many types of rail components and characteristics. This includes obvious, easy-to-notice defects like loose rail fasteners, rail cracks, rail burns, misplaced crossties, broken crossties, problems at joints, and defects of switches, as well as less visually evident defects like shifting from the abovementioned mathematical model over time. In
particular, a common problem in the railroad industry is the tendency of rails to deviate from their proper gauge.

The reasons for track defects are manifold. As to the rail defects, most of them develop over time due to wear. They occur due to the stress rails undergo from bearing heavy loads. The loads that the rails support, now typically in excess of 100 tons per wagon, create an outward pushing force that spreads the rails apart. The fasteners supply an opposing force to keep the rails in their intended location at the specified distance apart.

Aging and weather take their toll on rail components as well. Rails are constantly affected by forces of rail thermal expansion and contraction. Loose tie clips and/or defective crossties might lead to distortions in the track. In situations of extreme thermal variation, this can lead to a condition termed “sun kinks”, in which the track becomes instantly inoperable, sometimes during the course of one day or night.

A common cause of rail defects lies in the malfunctions of the wheels of a train. The segments of the rail on crossings and forks are especially prone to defects.

Since the avoidance of defects is so important, the rails must be inspected periodically, and maintenance is to be performed in timely manner should defects be discovered.

All of the aforementioned defects can either lead to accidents or cause other problems which increase the likelihood of accidents. Therefore, track operators make routine inspections of the tracks. It is their job to continually inspect the tracks and make any necessary repairs. Loose or missing clips and fasteners, faulty rails, joints, crossties, bolts, or spikes need to be repaired or replaced.
Since the replacement of track rails and other track details is a labor intensive operation, which is expensive and also impairs the railroad circulation, the correct moment for such replacement should be precisely determined, for example, by evaluating the periodic measurements of the rail parameters. The frequent monitoring also enables railway staff to decide when to remove speed restrictions which may be imposed after track maintenance, and indeed when to impose speed restrictions if track quality decreases. If track quality could be measured on a daily basis, this would also improve railway safety and decrease the risk of derailment.

In particular, it is well known in the railroad industry that there is a significant need to assure that the gauge of the rails meets certain minimum and maximum requirements to assure optimum travel along the rails while preventing derailment.

All railway companies are required to keep their track structures maintained to minimum safety standards with regard to track geometry. Two of these geometric standards are determined by measuring track alignment and profile. This is outlined in the United States of America Department of Transportation Federal Railroad Administration Track Safety Standards. These measurements are most commonly used by federal, state, and railway company inspectors as well as by track engineers and track constructors.

Previously, inspectors used to inspect the rail for defects visually or with the help of simple portable devices. Today, electric, magnetic, ultrasound, radar, lasers, and now computer vision methods may aid the inspection process of various components of the railroad.

In particular, it is now possible to extensively detect the above-mentioned defects and hazardous conditions by vehicle-based technological systems. The interest has grown over the years in the railway sector to deploy technological systems that can detect the presence of such defects and
hazardous conditions for rail vehicles. A principal reason for such interest is of course the intention to prevent accidents and losses (including indirect losses resulting from the interruption of a rail line). A further important motivation for the railways to deploy said detection systems is their interest in lowering the infrastructure and rolling stock maintenance costs that still constitute a major fraction of the total operation cost of the rail transportation systems. The large reduction that has occurred over the last few years in many countries of the manning of the railway infrastructure and of service sites has generally made said detection systems more useful, at least because the likelihood that defects and hazardous conditions are detected by personnel during inspections or by a casual observation has consequently decreased. Additionally, the reduction of maintenance budgets in certain countries or by some railway companies has resulted in a growth of the frequency of hazardous defects in the vehicles of the relevant fleets.

1.2 The research approach and organization of the thesis

From the discussion in Sections 1.1 and Related Work it is obvious that there is a gap between low-cost hand systems and complex industrial devices. We believe that this gap can be successfully filled by devices that use well developed and known methods of computer vision. We have observed that all applied systems sold today contain very little in respect of algorithmic load. Typically they contain a set of complicated hardware, but, as far as the algorithms go, they are often as simple as the use of the Pythagorean theorem and the like. In regards to this, the main goal of our investigation is to discover potential of computer vision methods in respect to this problem. A self-powered gasoline cart was built by us for data collection on railroads in the state of Florida. We used setups that use only cameras, setups that use only lasers, and setups using both;
we also tested setups that use laser triangulation approaches. The description of the hardware and experiment schematics resides in Section 1.3.

However, the main stress of this work is not put on hardware but rather on:

1) The creation of reasonable, simple, but reliable and fast computer vision algorithms, capable of real-time processing of big volumes of experimental field data and finding railroad defects reliably.

2) Comparative evaluation of performance of different algorithms and design schematics; uncovering their better and worse features.

3) Development of recommendations for future hardware development.

In particular, we have designed two systems that measure rail gauge using computer vision methods, one for laser scanners and one for cameras. We have tested both systems and found that the results are verifiable. The comprehensive comparison of both approaches as well as the comparison to the laser triangulation technique is within the scope of the dissertation work. Besides this, we show how computer vision methods can be applied to find other railroad defects (Section 5). This chapter will also be expanded and more railroad defects and results will be added. We understand the significance of doing all calculations in real-time. We show how finding railroad defects can be sped up and moved from Matlab into real-time domain by using novel hardware like GPUs. We dedicate the whole section to this.

The thesis contains 4 major sections. Section 3 discusses a method which finds a rail gauge based purely on laser range scanners, while Section 4 discusses the method which finds a rail gauge based on purely optical cameras. We show how computer vision methods can be applied to find other rail track problems in Section 5. Section 6 is dedicated to making all described
algorithms work in real-time. It describes ways to implement in whole or in part the above algorithms on the GPUs, parallel processors embedded in graphics cards widely available today.

1.3 The hardware used in experiments

For our experiments, we have built two self-propelled carts (Fig.1) which move over the railroad track. The two industrial laser range scanners are installed on a cart directly over each rail (Fig.2). The characteristics of the lasers are such that they scan 700 points each with angular step of 0.1 degree, which at the fixed distance to the rail of 2.2 feet corresponds to about 20 distance measurements (points) per inch. Given the standard rail width of 2 ¾ inches, this roughly corresponds to about 50 measurements on top of each rail head. The scanners are placed perpendicularly to the rail, and their co-alignment has been measured. Every scanner is able to see only one rail. They are able to produce more than 200 readings per second, which, at the average cart speed of 20 mph, corresponds to 5-6 readings per foot. According to industrial requirements, the systems of this type must produce at least one measurement per foot of rail track. It is typical for range scanner systems to have some uncertainty in measured signal (noise), which, for our setup, corresponds to about +/-0.5 inch of noise for every distance measurement. The typical scans of a rail head profile measured by scanners are displayed in Fig. 4, 5.
The same cart was used to acquire video sequences of the rail from two synchronous cameras installed directly over each rail. Some of those videos were taken with the help of artificial illumination (two strobe lights installed) while some others were not. The complete dataset of videos is documented in Section 4. Some of the videos were taken using installed shielding, which prevented direct sunlight in the images. This feature is important for removing shadows and glare from the picture. Some videos were taken with cameras installed on the side of the cart; in this way sequences of rail spikes and rail plates were recorded.

The other type of experiments we performed were experiments with the laser triangulation approach. This method requires two installed cameras coupled with two lasers capable of projecting a laser-induced line onto the rail surface. Prisms were installed on laser ends to achieve that (Fig. 3).
Figure 2: The schematics of the cart we used in the experiment. The two lasers are installed over each rail perpendicularly to the rails. The fields of view are known to move to further than $\pm 4/\pm 9$ degrees in every direction, so the camera views are reasonably constrained.

Figure 3: Laser triangulation method in its essence. a) The laser projects a line onto the surface of the rail, which, due to the rail geometry, is split into several lines on fields of view of camera. b) The projected laser line after the processing stage. The edges of the line define the edges of the rail.
Figure 4: The profile of a single range scan acquired by the device in raw unaltered data. In blue, the supposed position of the rail is overlapped on the image. In red we highlight points in the vicinity of the rail. The device possesses intrinsic error in distance measurements of about +/-0.5 inch.

Figure 5: Rail head close-up in the input data.
1.4 Summary

This section contains a brief introduction to the dissertation topic. In its first part, we give an overview of the railroad components, their parameters, and geometry, give the broad definition of a track defect as well as the causes of defects, stress the importance of timely detection of said defects which is crucial to support the safe operation of railroads, and give a short account of traditional ways of defect detection. In the second part of the introduction we describe the organization of the dissertation, set our goals, and point out the main priorities. In the third part of the introduction we give a short description of hardware (carts, lasers, and cameras) used in our experiments.

The next section contains the literature review of optical methods for track defect detection; we give special attention to visual methods.
CHAPTER 2
LITERATURE REVIEW

There are many principally different approaches to railroad track inspection (a concise overview can be found in [Tho95]). Technologies include: electromagnetic inspection methods, such as the alternating current field measurement [TS05], magnetic particle inspection or eddy current testing for rail crack detection, ultrasonic techniques for inspecting rail inner structure [HC95], ground-penetrating radars for inspection of the underlying ballast [KHS01], and methods for evaluating wheel acceleration anomalies for locating track wear [CCR07].

2.1 The optical methods

Among all other methods the optical methods stand apart.

Optical methods offer advantages over some other techniques for a few reasons:

1) They are non-invasive. Some other approaches are hindered by rail lubricant and require the lubricant to be removed so the tool will give accurate results.

2) They are precise.

3) They offer the possibility to measure many parameters of the track simultaneously.

The simplest devices in this category are called laser gauges. The simplest optical instrument of this category is called the laser gauge, which is commonly used in the railroad industry for measuring geometrical characteristics of railroad tracks. The laser gauges have been designed to facilitate railroad track alignment and/or track profile measurement. The laser gauges, which are used to measure the distance between the rails (gauge), can be mounted upon a railroad car and
propelled along the track to be inspected. The laser gauges are operated to accurately sense track defects, variations in track profile, and other track irregularities which might result in dangerous conditions.

Typically, operators utilize separate laser gauges and cross-level sensors for the measurement of the railroad track. The most commonly used cross-level sensors are solid bars connecting the left and the second rails. The bars are about 59 inches long, which can barely be fit into the trunk of the car. Therefore, most people carry a tape measure instead of the gauge, and ignore the cross-level measuring. The up-to-date, very compact (handheld) integrated device that includes laser gauge, cross-level, and microprocessor has been developed by Ensco. Inc. [US Patent 7,394,553 “Integrated measurement device”, 2008].

2.2 Laser triangulation methods

As the opposite to such simple devices as laser gauges, track vehicles with complex optical systems for measuring many different attributes of a railway track were used for track inspection as early as the 1970s.

One such system was the Optical Rail Inspection and Analysis (ORIAN) [Mag95]. It gave rise to a class of rail profiling systems which used triangulation techniques. This is a combined approach which uses cameras as well as laser diodes. Laser diodes project a stripe of light onto the rail which is visible to the camera. Since the relative positions of diode and camera are known, it is possible to recover the 3D rail profile by triangulation. Another system of this sort was developed by Range Vision, Inc [Bac95] and yet another similar system was created for the Moscow metro [RPS99]. The challenge for this class of systems is to segment out the projected laser line from the image, since the daylight scene illumination is much brighter than safe-to-use
low power semiconductor lasers even if cameras are equipped with narrowband pass filters of 10nm width or less, working at laser’s frequency. While indoor applications like Metro will have no problem, outdoor applications require shielding from direct sunlight. If this is possible, laser triangulation systems are able to provide good rail profiling accuracy at high speed. To our knowledge the best results in this area were achieved by Technogamma S.p.a. [http://www.technoeurope.it] and Ensco, Inc. [http://www.ensco.com]. The ‘Track Geometry Measurement System’ of Ensco allows real-time measurements of many track parameters under actual traffic loads and under all weather conditions.

However, those systems, despite their obvious advantages, possess a common deficiency – they are quite expensive both to purchase and to operate, therefore railway maintenance staff can inspect track lines only at infrequent intervals, which is typically once a month on the busiest lines and less frequently on others.

2.3 Visual methods

Far fewer published works use computer vision to look for rail problems. In particular, very limited literature exists combining camera platforms and computer vision techniques to detect rail defects.

These methods are cost-effective and simple, since they involve only the use of one or two cameras. Computer vision is suitable for railroad tracks monitoring, since railroad imagery is reasonably constrained (i.e. they contain mostly fixed-form man-made objects which can be detected with vision methods). Since cameras typically look downwards there are very few unexpected objects in the camera’s field of view; illumination and shadow patterns are also predictable. The more complex situations, like railroad crossings and forks, still pose problems,
since there are very many different configurations of crossings and forks. Today’s computers, contrary to some beliefs, cannot see, since seeing, or the ability to evaluate and distinguish between different situations including situations unknown before, implies intelligence. Vision algorithms are thus typically designed to work under specific conditions only, but they may fail to verify correctly if this condition is present. Thus, the development of practical vision systems for track inspection is complicated because it is difficult to make such systems perform correctly 100% of the time. The potential of vision systems is reviewed in [MF98].

Single camera vision systems are not suitable for detecting rail geometry since it is difficult to acquire depth from a single image. However, depth information can be easily obtained from stereo camera systems or complementary devices, such as range scanners. The approach of [Uka96] uses stereo cameras to detect obstacles on the track.

Paper [MNA04] discusses the applicability of texture analysis methods to detecting rail corrugation. Gabor filters, wavelet, and Gabor wavelet transforms are compared to determine which transform combined with a support vector machine learning approach detects corrugation best. Also, several recent papers by Mazzeo and Stella [MNS04, MSN05, SMN02] discuss detecting missing rail bolts with the use of a combination of the wavelet transforms, principal component analysis, and trained neural networks. Their methods assume the distance between bolts is constant and known a priori. Bolt detection/classification is improved by simply looking in the expected location of a bolt.

Another work of interest uses line-scan cameras and spectral image differencing procedures to find mechanical defects on the top of the rail including tiny cracks, flakes, grooves, or break-offs [DGN04]. Two additional light sources (red, blue) are used intermittently; on non-flat surfaces
the amount of reflected light from one light source is increased and the amount from the other light source is decreased.

In a recent work, a vision-based approach is used for finding cracks on rail joint bars [DOT06].

In our laboratory a method was recently developed for monitoring obstacles on railroad crossings [SZS04]. This method exclusively uses vision techniques - background subtraction, object detection, and tracking - to detect objects which are moving in close proximity to the tracks ahead of an incoming train.

At the time of publication, there were no papers published which measure rail gauge using purely vision methods, that use statistical, normalized cross-correlation and MACH filters, or that detect rail burns using vision methods.

2.4 Summary

The literature review has uncovered that, while there exists a general understanding of advantages of computer vision techniques for railway defect detection, not much work has been done yet, especially for the detection of a broad scope of defects (as compared to detection of rail gauge only), or the detection of defects in complex conditions, for example, in presence of railroad crossings or forks. The advantages and disadvantages of different computer vision approaches for both data acquisition (cameras, lasers) and data processing were not studied. The analysis of the very important problem of implementing computer vision techniques in real-time is missing from literature completely.

With the goal to fill these gaps, we are investigating both theoretical and practical aspects of the use of modern computer vision approaches in order to locate the defects in railroad tracks. These are discussed in detail in the following sections. In particular, Section 3 discusses a method
which finds a rail gauge based purely on laser range scanners, while Section 4 discusses the method which finds a rail gauge based purely on optical cameras. We show how computer vision methods can be applied to find other rail track defects in Section 5. Section 6 is dedicated to making the described algorithms work in real-time.
CHAPTER 3

DETERMINING RAIL GAUGE USING PURE LASER APPROACH

3.1 Introduction

We have developed a method to measure a rail gauge using two laser range scanners installed on a self-moving cart as discussed in Section 1.3. The algorithm to find the rail gauge consists of several steps outlined in Scheme 1. In general perspective, we determine the inner edges of both rails in scanner views and, knowing the distance between scanners, calculate the distance between rails in inches. We start with building a 3D panorama of the rail out of a stack of input scans (step 1). After the panorama is built, we apply FIR circular filtering and Gaussian smoothing to the panorama buffer to suppress the noise component (step 2). In the next step (3), we attempt to segment the rail heads in the panorama buffer. We employ the method (step 4) which detects railroad crossings or forks in the panorama buffer. If they are not present, we find the rail edge using robust line fit (step 5). If they are present we use an alternative way: we predict the rail edge positions using Kalman filter (step 6). In the next, 7th step, common to both fork/crossings conditions, we find the adjusted positions of rail edges using additional (mean shift) clustering in the vicinity of the edge. We approximate the rail head surface by the third degree polynomial and then fit two plane surfaces to find the exact position of the rail edge. Lastly, using rail edge information, we calculate the rail gauge and smooth it with 1D Gaussian filter.
Scheme 1: The solution diagram to railroad rail edge detection and gauge measurement.
3.2 Algorithm details

3.2.1 Building 3D panorama of the rail

Out of a sequence of input scans we can stitch a panorama buffer of variable length (typically 30-50 scans). The scale of the panorama buffer in the temporal dimension is determined by the hardware encoder, the results of which are in turn improved by a built-in high precision GPS unit. The encoder counts the number of cart wheel rotations and thus presents a measure of distance along the track with a relative precision of about 1%. If spatial geometry of the track is known beforehand, this distance measurement is further improved by the readings of the GPS unit. For our experiments, at average cart speed of about 30 mph, we obtained about 5-6 readings per feet of track; thus the intrinsic property of the measurements is that the density of measurements in the temporal dimension $Z$ is about 50 times less that in the $X$ dimension. The Fig. shows an example of a panorama buffer for one (left) rail.

3.2.2 FIR filtering and Gaussian smoothing of input data

Due to the intrinsic noise and scanning errors in the input data, the first, necessary algorithm step is filtering and smoothing the data set. We are applying two different algorithms to filter the input data sets. Both algorithms are applied to the panorama buffer. As the first step, we are applying the low-pass Finite Impulse Response (FIR) [Stu96] filter to remove high frequencies from the input data. The filter general form is written as

$$h(n) = \frac{1}{N} \left[ A_0 + \sum_{k=1}^{M-1} 2A_k \cos \left( 2\pi (n - M) \frac{k}{N} \right) \right].$$
where $A_0$ thru $A_k$ are the desired frequency samples, $N$ is the number of samples and $M$ is the median value $M = (N - 1) / 2$.

We are using a 2D (circular) version of the filter which we apply in the XZ dimension. The radius of the working window is set to 5-7 data points. We are not applying a filter to the points in the working window for which we observe a significant change of Y value compared to the Y value of center point; thus the surfaces of the track with different heights (like rail heads and ground, for example) are filtered separately.

Since the input data contain some incorrect points which do not represent actual distance readings but rather errors from the hardware, we need to locate and remove erroneous points. We are locating outlier points by using a sliding window in the XZ temporal dimension. The mean $M$, standard deviation $S$, and variance $V$ of $P_Y$ coordinates in the window are calculated. Points are marked as outliers if $|P_Y - M| > S$. These points are removed from the input sample.

As the last filtering step we are applying 2D Gaussian Filter in the temporal dimension XZ:

$$G(x, y) = \frac{1}{2\pi\sigma^2}e^{-\frac{x^2+y^2}{2\sigma^2}}$$

where $\sigma$ is a standard deviation of Gaussian distribution. We apply the same restriction for the Gaussian filter as for the frequency filter: we do not include points into filter calculation if their deviations of the Y coordinate exceed the standard deviation.
3.2.3 Segmenting out the rail surface by using k-means clustering

Our next step is to find a rail surface in the filtered panorama buffer. The scanners on our cart are placed directly over the rail (Fig. 2). In this setup, the rail heads always form separate point clusters in data (Fig. 5). This happens due to the intrinsic rail geometry. We are applying 3-means clustering algorithm to points in the panorama buffer.

Due to the geometry of the rail and experiment, all data points are roughly grouped around 3 clusters: clusters $S_1$ and $S_2$ are for points to the left and right side of the rail, while cluster $S_3$ is for the points on the rail head. The typical example of such clustering is shown in Fig. 6. To separate the rail head cluster $S_3$ from the other two point clusters we used iterative k-means clustering algorithm [KMN02].

The beginning centroids of clusters $\mu_j^{(0)}$ were selected heuristically; in particular we believed that the centroid of $S_3$ is at the expected location of the rail head. All points were attributed to one of three clusters $S_1^{(0)}, S_2^{(0)}, S_3^{(0)}$ based on the proximity to the cluster centers. The cluster centers are then re-computed ($\mu_j^{(0)} \Rightarrow \mu_j^{(1)}$) and all points are attributed to one of three new clusters $S_1^{(1)}, S_2^{(1)}, S_3^{(1)}$. The cluster centers are recomputed ($\mu_j^{(1)} \Rightarrow \mu_j^{(2)}$) and the process starts over. We calculate intra-class variance on the each iterative step:

$$V^{(n)} = \sum_{i=1}^{3} \sum_{x \in S_i^{(n)}} \|x - \mu_j^{(n)}\|^2$$

The process ends when the change in $V$ is negligible ($V^{(n+1)} - V^{(n)} / V^{(n)} < 10^{-3}$).
3.2.4 Verification of presence of railroad crossing or fork

The algorithm described in step 3 works fine for all cases where the rail head is easily separable from the other objects in picture. However, there exist special conditions which must be taken care of: railroad forks or railroad crossings may be present in the picture. If a railroad fork is present then there is more than one rail head in the scan, and for both rail fork and rail crossing conditions the ground is level with the rail heads, which makes rail heads inseparable from the ground. There is, however, a small groove on the ground where cart wheel edges fit in (Fig. 7), which is always present. The general solution to this situation is rather not to use k-means clustering but instead to track the positions of rail edges found in previous scans. In order to properly handle such situations we must first employ a method that detects unusual conditions in the scans. We make use of a low-width 1D sliding window which examines every input scan for the presence of unusual conditions. The responses from the scan procedure are given as the
probability scores; the higher the score the higher the probability that the scan contains the forks/crossings.

By combining the responses from neighboring scans using Bayesian transformations we estimate the chance that the scan \( n \) contains the forks/crossings. This approach is discussed below.

For the point of view of probability theory the results of the scans for the presence of an unusual situation (crossing) form the discrete-time random chain, where each node of the chain has two states, \( X_n = 1 \) (scanning finds the crossing) и \( X_n = 0 \) (scanning does not find the crossing).

Those two states (0 and 1) do not imply that the crossing does actually exist – they imply that the crossing was detected by the hardware/software, and no more than that.

Figure 7: Railroad crossing in raw input data. The small groove is visible to the right of the railroad rail. The railroad truck wheels fit into the groove, so it must be always present. This groove is hard to detect by standard methods, so it needs special processing by Kalman filter prediction mechanism.

**Step 1.** Let's consider the simplest case, when we know the width of the crossing and the speed of the measuring device, that is, the number of scans \( L \) that belong to the crossing. Our algorithm that detects crossings switches on after we detect some \( X_n = 1 \) state after a long sequence of \( X_n = 0 \) states. Therefore, we consider only sequences of states of length \( L \) that start with 1. Let’s call an instance of this sequence, an *event B*:
\[ B = \left\{ X_N = 1, X_{N+1} = \ldots, X_{N+\ell} = \ldots \right\} \]

while we call an *actual* presence or absence of a crossing \( A_0 \) and \( A_1 \) events, respectively (again, true only if \( X_N = 1 \) holds).

Let’s introduce two transition probability matrices:

\[
\begin{bmatrix}
p_{00}^{(0)} & p_{01}^{(0)} \\
p_{10}^{(0)} & p_{11}^{(0)}
\end{bmatrix}, \quad
\begin{bmatrix}
p_{00}^{(1)} & p_{01}^{(1)} \\
p_{10}^{(1)} & p_{11}^{(1)}
\end{bmatrix}
\]

The upper indices \((0)\) and \((1)\) refer to the events \( A_0 \) and \( A_1 \) respectively. The matrices encode a transition from one state to another while we go from one scan to another in an input sequence:

\[
p_{ij}^{(0)}(n) = P(X_{n+1} = j \mid X_n = i \text{ and } A_0), \quad p_{ij}^{(1)}(n) = P(X_{n+1} = j \mid X_n = i \text{ and } A_1)
\]

Those 8 probabilities are in fact experimental values (priors) which depend on both hardware and crossing properties. They can be derived from the experiments (training) and prior knowledge of crossing types. We have to stress that those probabilities refer only to \( B \) events, rather than the whole chain of events in sequence. If hardware and software characteristics are good enough then \( p_{01}^{(0)}, p_{11}^{(0)}, p_{00}^{(1)}, p_{10}^{(1)} \) are close to 0, while \( p_{01}^{(0)}, p_{11}^{(0)}, p_{00}^{(1)}, p_{10}^{(1)} \) are rather close to 1. We note that there might exist a dependence of transition probabilities on scan number \( n \), which is seen from formula (3). For example, we see that typically \( p_{11}^{(1)}(N) < p_{11}^{(1)}(N+1) \), since while response \( X_N = 1 \) can be caused by noise rather than actual presence of a crossing, two noise responses in a
row are not highly probable. If we assume that there is no dependence of event $B$ on $n$, then we are dealing with Markov chain model here.

**Step 2.** Given Markov chain model, we can compute the conditional probability of any particular sequence of states $B \quad 1, j_{N+1}, j_{N+2}, ..., j_{N+L}$, where $j_k = 0,1$, using multiplication rule:

$$P(B \mid A_i) = p_{1j_{N+1}}^{(i)} p_{j_{N+1}j_{N+2}}^{(i)} ... p_{j_{N+L-1}j_{N+L}}^{(i)}$$

**Step 3.** Using Bayesian formula we can calculate the probability of the event $B$ being the actual presence of a crossing (event $A_i$):

$$P(A_i \mid B) = \frac{P(B \mid A_i)}{P(B \mid A_i) + \frac{P(A_0)}{P(A_i)} P(B \mid A_0)}$$

where $\frac{P(A_0)}{P(A_i)}$ is a ratio of probability that a crossing exists to the probability that it does not exist (given $X_N = 1$). This ratio can be obtained from the experiment by counting the number of false positives in the system.

If $L$ is not known beforehand, we can determine it empirically. For example, we can define that if after a chain of several 1s and, possibly, few single 0s, we see two 0s in a row, those two 0s define the end of the sequence.
3.2.5 Finding the rail edge if forks/crossings are not present (Robust Line fitting)

If forks or crossings were not found in the scan, we fit a line into each scan using the robust line
fit approach. We minimize an error function:

\[ \sum \ln(1 + \varepsilon^2 / 2) \]

This variant of error function does not generate significant errors for the outlier points due to the
use of natural logarithm. We fit a line into each scan first in the Y, then in the X dimension. The
line fitting gives us a good estimate for the centroid of the rail head, the Y coordinate of which
matches that of the rail edge. We extend the line towards the rail edge and proceed to step 7 to
find the exact position of the rail edge.

3.2.6 Finding the rail edge if forks/crossings are present (using Kalman filter
prediction scheme)

If forks/crossings are present in any scan in the panorama buffer, we have to use an alternative
approach to find the inner rail edge in that scan. We are employing the discrete Kalman filter
prediction scheme for that. The prediction is based on the previous measurements of rail edges as
well as an edge movement linear model with Gaussian noise distribution. In the Kalman filter
model [Kal60], the true model state (time k) is derived from the state at time k − 1 according to

\[ z(k) = \Phi(k, k - 1)z(k - 1) + w(k) \]
where $\Phi_k$ is the state transition model and $w_k$ is the noise, which we assumed to be Gaussian. At time $k$ an observation $y_k$ of the true state $z_k$ is made according to

$$y(k) = H(k)z(k) + v(k)$$

where $H_k$ is the observation model which maps the true state space into the observed space and $v_k$ is the noise which we assume to be Gaussian white noise. We assume a steady model state $\Phi_k = \Phi_{k-1}$ and constant velocity model for $v_k$. The initial state and the noise vectors at each step are all assumed to be independent. On every Kalman filter iteration we update system and measurement models and find predicted positions of the rail edge. We are substituting Kalman filter-predicted rail edge positions for every scan where forks/crossings were found.

The algorithm works for every new laser scan, which we number 1, 2 … etc. It works iteratively for some fixed number of iterations for every scan.

The input parameters are:

$\bar{x}_0$ - a priori estimation of rail edge position (obtained from data). If this is the first scan on rail crossing, we take the rail edge position linearly interpolated over the last 20 scans. If this is not the first scan on the crossing, we take the result of linear interpolation of previous rail edge positions taking into account Kalman filter adjustments.

$P_0$ – the initial value of error variance. It should not be equal to 0. We assume $P_0 = 1$.

$Q$ – is a process variance. It depends on iteration number in general case. We assume it to be

experimental constant($10^{-3}$).

$R$ – is a measurement variance constant. We set it to 0.1.
The first iteration step is updating a priori estimate:

\[ \bar{x}_k = \bar{x}_{k-1} \]
\[ \bar{P}_k = \bar{P}_{k-1} + Q \]

The second iteration step is a posteriori estimation (Kalman gain):

\[ K_k = \frac{\bar{P}_k}{\bar{P}_k + R} \]

Update estimate with measurement:

\[ x_k = \bar{x}_k + K_k (z - \bar{x}_k) \]

Update the error variance:

\[ P_k = (1 - K_k) \bar{P}_k \]

Convert a posteriori estimates into a priori for the next iteration step

\[ x_1 \Rightarrow \bar{x}_1, \quad P_1 \Rightarrow \bar{P}_1 \]

And transition to the next iteration \( k=2 \).

3.2.7 Find adjusted positions of rail edges using mean shift clustering

To refine the rail edge predictions given to us by Kalman filter model (if forks/crossings are present), or to refine rail edges detections from robust line fit (if forks/crossings are not present), we are running an additional clustering algorithm which clusters points in the immediate vicinity of the edge predictions using mean shift clustering approach. We define a window of certain size around each predicted edge point and find the rail extents in this window using mean shift
clustering. As with most of our algorithms, we run it in the XZ plane. The edges of the found cluster define the improved edges of the rail.

3.2.8 Approximate rail head surface by the third degree polynomial

The result of execution of the previous seven steps is a set of rail edge points, one for each scan line. However, the positions of those points can contain some measurement errors in themselves. On the other hand, we know that the rail edge is a straight line, therefore the rail edge points must be located on the same line. Therefore, we employ a two-step approach to enhance the rail edge position detections. We fit the surface approximated by the third degree polynomial into the vicinity of edge points in the panorama buffer. After this step, we fit a vertical plane tangent to this surface. The intersections of this plane and the scan lines define a line all rail edge points must rest upon.

After we determine the inner rail edge points for all scans in the panorama buffer, the last, fairly straightforward step is a calculation of rail gauge between matching points. Since the two laser scanners have already been aligned, the distances between matching edge points are calculated and stored as a rail gauge. We are applying post-processing to the rail gauge curve by smoothing it with 1D 1x5 Gaussian filter.

3.3 The method of evaluation

The experiments were performed on data obtained from visiting different track locations in Central Florida. During the experiments, we accumulated over 1000GB of laser range scanner measured profiles of railroad tracks; those profiles were synchronized for both rails. The experimental validation was performed in-situ by measuring actual gauge and comparing it to
calculated values. The indirect algorithm validation was performed by running a railroad cart over the same segments of track more than once and comparing the obtained gauge curves. We have observed that gauge curves overlapped in this way always closely match (Fig. 8). The precision of the experiment was found to be consistent with precision of hardware used in the experiment.

![Figure 8: The profile of a rail gauge measured in inches at a track segment of about 8000 feet long. In the experiment, we ran the cart over the same track more than once. We overlap the resulting gauge curves on this figure. We observe that overlapped gauge curves closely match.](image)

The detailed evaluation of the algorithm on the whole set of 1000GB of data as well as a qualitative comparison to both other major algorithms, camera-based and laser triangulation methods, was performed. The current drawback of the algorithm is that the real-time processing speed is limited: the MATLAB implementation results in about 10-15 scans per second while the scan incoming rate from the device is about 400 per second. At the regular cart speed of 20 mph, the required processing rate is at least 30 scans per second. The implementation of the parts of the algorithm using the GPU library and GPU techniques we developed (discussed in Section 6) can be a topic of further study.
3.4 Summary

In this section we have shown how vision methods can be applied to detect parts of railroad tracks based on input from a pair of industrial laser range scanners. We have developed and tested a computer vision method to detect parts of railroad tracks for data represented as a point cloud as read by a laser range scanner. We described in detail the steps of the algorithm based on vision techniques (robust segmentation using k-means and mean shift clustering, Kalman and FIR filtering, 3D line and plane fitting), including a check for possible crossing and forks presence. The results were validated on the real data sequences from the railroad tracks as well as the artificial data generated in the laboratory installation. A series of real-time experiments on railroad tracks were performed, which have always shown consistent results.
CHAPTER 4

DETERMINING RAIL GAUGE USING PURE CAMERA APPROACH

4.1 Introduction

A self-powered battery cart was built for data collection. The cart, pictured in Fig.1, has four wheels which are in direct contact with the rail, as well as four smaller directing wheels. Two high-speed CCD cameras are simultaneously collecting data from the two rails and the immediately adjacent areas. Strobe lights synchronized with cameras are used to minimize the difference in illumination at different times of day and varied lighting conditions. Sun shields were installed to remove the shadow component from images in data collections. The computer, GPS unit, and wheel encoder were also installed on the cart.

This section deals with some of the more conventional computer vision methods. It is meant as a demonstration of computer vision methods for rail defect detection. In the following section we will present an applied system for the real-time measurement of the rail gauge.

The rail gauge is the distance between the innermost sides of the two parallel steel rails. Rail gauge is the most important measurement for track maintenance, because deviations in gauge indicate where potential defects may exist. Rail gauge standard varies by country; in the USA it is 56 ½ inch. If the gauge at any given location in the USA deviates below 56 or above 58 inch the train could potentially derail. Therefore, the gauge measurements, as well as the visual track inspection, are the primary means of uncovering potential defects. Portions of the rail with loose or broken fasteners, damaged or misplaced crossties, or cracked rails all typically show deviated gauge.
We present an algorithm which locates the rail in the image and estimates the rail gauge from synchronously taken images of parallel rails. The first step in measuring rail gauge is to accurately detect the rail in each of the two non-overlapping synchronous images from the two cameras installed on the data collection cart. The cameras are placed right over the rails at an almost minimal focus distance of about 2 feet, which allows it a detailed view of the rails. With the rail location detected in both synchronized cameras, the rail gauge is a straightforward measurement, assuming that cameras were calibrated. Therefore, our approach focuses on accurately detecting the rail in images. Camera calibration estimates the camera’s relative positions as well as the scale of the objects in each camera and is discussed in Section 4.5.

### 4.2 Detecting the rail in the image

The rail consists of two main parts: the rail head, which the train or cart wheels lay directly on, and the rail base, which is attached to crossties with tie clips or fasteners (Fig. 9). Since the gauge is measured between rail head edges, when we mention detecting the rail it means detecting the rail head. Our testing vehicle is constructed so that the cameras look directly down onto the rail. The rails are not necessarily oriented vertically in the images. Since the cart uses rubber wheels which slip during movement, the rail orientation in the image can deviate from its upward direction to about 5-10 degrees; thus it appears that the rail is moving within the image.
Figure 9: The profile of the rail. The topmost part the wheels lay on is the rail head, while the bottom part attached to the ground is the rail base. According to US standards, the rail base is always about twice wider than the rail head.

Rail detection consists of three main steps:

1. Build an edge map and fit lines into an edge map using the Hough transform
2. Detect persistent edge lines using a history buffer
3. Detect and validate the rail

Building edge maps. Edge maps are generated from the input images with a Sobel filter. Typically, vertical and horizontal Sobel filters are used in edge detection. Since our goal is to find the rail, we only use the filter that detects vertical edges parallel to the rail (Fig. 10a). This edge map is then used as input to a Hough line transform [SS01]. An intermediate result is shown in Fig. 10b.

Looking at the rail above, up to four distinct parallel lines may be visible. These four lines are made up of the two lines that define the rail head which lie within the two lines that define the
rail base. The intrinsic property of these lines is that they are always parallel to each other due to the structure of the rail. Due to the way our cart is set up (discussed in Experiments section), these four lines always generate high response edges in the edge map. Since rail lines run through the entire frame we can assume with high probability that the longest line found by the Hough space matches one of the four rail lines. The longest line is found at a maximum in a Hough space.

In Hough space, all lines that are parallel to each other are located in the same column (Fig. 11). Once the longest line in Hough space is determined, its orientation is used to eliminate lines that differ significantly in orientation. We refine the list of remaining lines by removing short lines and suppressing lines which appear in close proximity to each other.

![Figure 11: A Hough transform of a Sobel edge map. All lines in Hough space which are co-parallel with the longest line in Hough space are located in the same column. We allow for some margin by including a few adjacent columns. We eliminate all other lines. Red rectangle in picture shows the (approximate) position of the cut-off in Hough space.](image)

Detecting persistent lines. The previous step detects candidate lines that are parallel to the rail. However, these lines may not be rail lines. Sometimes objects in the scene, such as rail plates, sleepers, or fasteners, can produce false candidate lines (Fig. 12). A history buffer is created to hold information about the detected candidate lines. As the cart moves, candidate lines that persist are considered rail lines. These non-rail objects only appear periodically. The ballast,
typically made up of rocks, changes continually and has no persistent linear pattern. The sleepers and fasteners are not present in the majority of the images. Using this history information, the most persistent lines are likely to define the rail (Fig.13 a).

Figure 12: The tie plate which appears in this image to the right side of the rail generates strong edge close to parallel to the rail edge.

*Detecting and validating the rail.* The problem is to determine which two of the remaining candidate lines define the rail head. The distances between these detected lines are used to determine the type of lines represented. The cameras on our cart are set up so that the distance in the image plane between the edges of the rail head (yellow line on Fig. 13b) is always greater than the distance between an edge of the rail base (green line on Fig. 13b) and the edge of the adjacent rail head. Given a line, the distance is measured between the two neighboring parallel lines. The largest distance defines the rail head (Fig. 13b).

Figure 13: a) The candidate rail lines (green) are determined on step 2. b) The lines corresponding to rail head edges (yellow) are located in step 3.
Since not all frames will detect the four most typical rail lines, or other non-rail lines may be detected, a means of validating the two lines is added. The measured width of the rail is assumed to be mostly constant since it can change only at rail joints. This width is measured and recorded in pixels from frame-to-frame. We use a history buffer to track the rail width. If the rail head’s width measurement significantly deviates from the mean of the history buffer, its confidence is reduced (Fig. 14).

Figure 14: This figure shows an occurrence of different measured rail widths in a sequence of 200 images. X axis plots rail width, while y axis plots an occurrence. About 10 images produce significantly deviated rail width (represented by the second small spike on a figure). Those images are invalidated.

4.3 Estimating the distance between rails

Next we present an algorithm for determining the rail gauge from the data collected with synchronized cameras. The schematics of our setup are given in Fig. 15. Here, the $D_{\text{cam}}$ is a world distance between zero points in cameras (in world distance units, ex: inches). We find both zero points and distance $D_{\text{cam}}$ from camera calibration discussed in the following section. We also learn from calibration camera zoom ratios, $Zoom_L, Zoom_R$, which are the ratios of image–to–world objects sizes for objects seen at a rail distance. For every pair of input images, we measure the rail offsets, $Off_L, Off_R$, of inner rail edges from camera zero points along the axis
perpendicular to the rail in the image (Fig. 16). Based on these measurements and the camera parameters, we find the rail gauge \( G = D_{\text{cam}} + Off_L \cdot Zoom_L + Off_R \cdot Zoom_R. \)

Figure 15: This schematic shows how we obtain rail gauge \( G \) from estimates of rail positions in images. Arbitrary zero points on the \( x \) axis are chosen for each camera. The world distance between the zero points and camera zoom levels \( Zoom_L \) and \( Zoom_R \) are measured during camera calibration. Zoom levels define a pixel-to-inch ratio for objects in camera view at a rail distance. Rail edge offsets from zero points \( Off_L \) and \( Off_R \) are measured for every image. Thus, the rail gauge in world coordinates is found as \( G = D_{\text{cam}} + Off_L \cdot Zoom_L + Off_R \cdot Zoom_R. \)

Figure 16: Measured rail offset is an offset of an inner rail edge from the (right) camera zero point.

4.4 Results

We used a self-powered battery cart to perform experiments on a railroad track. The cart was equipped with a computer, which allowed us to perform all experiments \textit{in situ}, and two high
shutter speed cameras. In order to remove the shadows and lessen the midday sunlight interference, two conic shields were installed on the cameras, one per camera. We used two strobe lights synchronized with the camera shutters. Strobe lights were installed directly over the rail (at about 3” distance from the rail head and 8-10” distance from the rail base accordingly). This allowed for overexposing the rail head while the rail base got substantially less strobe light, which in turn generated strong edge boundaries between the rail head and rail base. We would like to mention that our experiments with rail detection without the strobe lights were unsuccessful, since the edges between rail head and rail base were either not prominent or completely absent from the images. This happened because the rail head and base have similar surface textures. The only possible solution for this problem without the use of a strong illumination source is to install the cameras at such angle that the inner edge of the rail head is projected onto ballast rather than the rail base. Ballast has vastly different texture properties from a rail head.

The use of camera shields was also essential for two reasons: the shadows generated by rail heads run parallel to the rail and generate strong false edges, and it is hard to dynamically select appropriate camera shutter and gain since there are significant variations of daylight illumination. In shadow areas the images tend to be too dark, while the midday sun easily overexposes the image. The combination of artificial illumination and shielding the cameras was also good in that it created a predictable artificial setup in which lighting condition did not change much from image to image. Images were thus not influenced by conditions external to the cart.
4.4.1 The evaluation data set

In experiments we used our own data set, which we gathered during visits to different railroad sites in Central Florida. All the videos in our data set are presented as 24-bit greyscale color in 320 by 240 pixels resolution. Some videos also exist in RGB color with different resolutions. The videos in the data set are grouped into 10 collections by video type. Table 1 summarizes all the data collections in our dataset, while Fig. 17 shows a collage of one video from each collection. Collections 8 and 10 contain videos of tie clips/e-clips/bolts, which hold rails to concrete crossties. They are useful for validating methods which account for missing/damaged tie clips/e-clips or missing bolts. Collections 5 and 7 contain videos of tie plates/spikes, which hold rails to wooden crossties. These videos can be used for similar purposes. Collection 6 contains many close-up videos of rail heads. We are using this collection to validate the rail detection algorithm in the following subsection. This collection can also be used to validate methods which look for missing tie plates and spikes. Collection 9 includes a video of wooden crossties taken from a 45 degree angle perspective. It can be used to detect misplaced crossties. Experimental collections 1-4 contain videos of rail/tie plates/spikes taken from different angles of view and camera zooms. These collections typically do not feature strobe lights or shielding. However, they are available in color and a higher image resolution.
Figure 17: A mosaic of images from each of 10 collections in our dataset.
Table 1: Our dataset we used in experimental validation.

<table>
<thead>
<tr>
<th>Collection</th>
<th>Video</th>
<th>Video type</th>
<th>Frame rate/ s</th>
<th>Frames</th>
<th>Strobe Light</th>
<th>Shield</th>
<th>Available in color</th>
<th>Color resolution</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>rail and spikes</td>
<td>15</td>
<td>300</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>320x240</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>tie plates/spikes</td>
<td>15</td>
<td>300</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>1024x768</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>rail and spikes</td>
<td>8</td>
<td>1300</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>1024x768</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>rail and spikes</td>
<td>8</td>
<td>1700</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>1024x768</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>rail and spikes</td>
<td>8</td>
<td>550</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>1024x768</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>rail and spikes</td>
<td>8</td>
<td>550</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>1024x768</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>rail and spikes</td>
<td>15</td>
<td>350</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>1024x768</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>rail and spikes</td>
<td>15</td>
<td>350</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>1024x768</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>rail and spikes</td>
<td>15</td>
<td>1350</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>1024x768</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>rail and spikes</td>
<td>15</td>
<td>550</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>1024x768</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>rail and spikes</td>
<td>10</td>
<td>550</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>-</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>rail and spikes</td>
<td>10</td>
<td>650</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>-</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>rail and spikes</td>
<td>10</td>
<td>700</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>-</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>rail and spikes</td>
<td>20</td>
<td>400</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>-</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>rail and spikes</td>
<td>20</td>
<td>800</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>-</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>rail and spikes</td>
<td>25</td>
<td>1700</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>-</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>rail and spikes</td>
<td>25</td>
<td>2500</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>-</td>
</tr>
<tr>
<td>6</td>
<td>8</td>
<td>rail and spikes</td>
<td>25</td>
<td>2900</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>-</td>
</tr>
<tr>
<td>6</td>
<td>9</td>
<td>rail and spikes</td>
<td>20</td>
<td>1700</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>-</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>tie plates/spikes</td>
<td>32</td>
<td>3200</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>-</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
<td>tie plates/spikes</td>
<td>30</td>
<td>1900</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>-</td>
</tr>
<tr>
<td>7</td>
<td>3</td>
<td>tie plates/spikes</td>
<td>30</td>
<td>1250</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>-</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>tie clips/e-clips/bolts</td>
<td>15</td>
<td>900</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>-</td>
</tr>
<tr>
<td>8</td>
<td>2</td>
<td>tie clips/e-clips/bolts</td>
<td>15</td>
<td>500</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>-</td>
</tr>
<tr>
<td>8</td>
<td>3</td>
<td>tie clips/e-clips/bolts</td>
<td>15</td>
<td>2350</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>-</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>tie clips/e-clips/bolts</td>
<td>15</td>
<td>950</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>-</td>
</tr>
<tr>
<td>8</td>
<td>5</td>
<td>tie clips/e-clips/bolts</td>
<td>15</td>
<td>1000</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>-</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td>crossties</td>
<td>20</td>
<td>550</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>1024x768</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>tie clips/e-clips/bolts</td>
<td>15</td>
<td>550</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>2</td>
<td>tie clips/e-clips/bolts</td>
<td>15</td>
<td>1600</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>3</td>
<td>tie clips/e-clips/bolts</td>
<td>15</td>
<td>850</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>4</td>
<td>tie clips/e-clips/bolts</td>
<td>15</td>
<td>6550</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>5</td>
<td>tie clips/e-clips/bolts</td>
<td>15</td>
<td>6650</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>-</td>
</tr>
</tbody>
</table>

4.4.2 Rail detection experiments

Since rail gauge measurement is straightforward as long as we locate the rail in images, we concentrated our experiments on rail detection. Rail detection performance was evaluated on dataset collection 6, which contains nine rail videos taken from different tracks. The total number of rail images in these videos is 11900. All videos share the same common properties. They were taken using both shield and strobe lights, the angle of view was upright, and the zoom level of the features in view was appropriate for our method. Some input videos were not fit for our experiments. In videos 3 and 7, the rail stays partially offscreen most of the time, which was an
incorrect initial condition for our method. In order for our method to work properly, the rail head must be fully visible all times. In video 8, the rail is staying outside of screen for about 20% of the time. We have decided to include video 8 in the testing set to show that we can work with boundary conditions. We have run our method on all the videos in the dataset, except 3 and 7, and present our findings in Table 2.

Out of 5800 input images (not including videos 3, 7, and 8), 485 were invalidated, which was 8.4% of the input. The first 10 frames of any sequence are automatically invalidated since it is the length of the history buffer. Many of the invalidated frames are located on the rail joints or litter accompanying the older rail tracks. At a frame rate of 15 fps and a cart speed of 10 mph, the cart is taking measurement about every 1 foot. This measurement rate is considered good for gauge measurement by railroad operators. Since the frames come at a rate of 10-20 per second, the invalidation of about 9% of frames (1-2 fps) is typically acceptable for an industrial gauge measurement system. We have to mention that there are cameras available which can produce significantly higher frame rates of about 60 fps and higher. Therefore, these are long sequences of invalidated frames where we can lose some gauge results. Our system included the feature that, if the gauge is invalidated, the gap in gauge data was displayed so the track operator could choose to verify the gauge manually.

34 out of 7500 validated images, or 0.39%, out of total 8700 (not including videos 3, 7) still produced incorrect rail detection. The worst incorrect detection rate of 2% was reported on short video 1, which got its validating history buffer initialized to the wrong rail width value at the beginning. It recovered after a few hundred images. However, incorrect rail edge detections with an occurrence of less than few percent can be easily dealt with by using a short history buffer
which tracks the rail edge positions and rejects outliers, and/or a smoothing operation on a rail edge position or rail gauge.

Table 2: Results of rail detection evaluation on data collection 6.

<table>
<thead>
<tr>
<th>Collection</th>
<th>Video</th>
<th>Frames</th>
<th>Validated frames</th>
<th>Invalidated frames</th>
<th>Percentage of invalidated frames</th>
<th>Incorrect rail edge detections</th>
<th>Percentage of incorrect rail edge detections</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>1</td>
<td>550</td>
<td>460</td>
<td>90</td>
<td>16.3%</td>
<td>11</td>
<td>2%</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>650</td>
<td>607</td>
<td>43</td>
<td>6.9%</td>
<td>4</td>
<td>0.6%</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>400</td>
<td>341</td>
<td>59</td>
<td>14.75%</td>
<td>0</td>
<td>0.0%</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>800</td>
<td>734</td>
<td>66</td>
<td>8.2%</td>
<td>2</td>
<td>0.25%</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>1700</td>
<td>1641</td>
<td>59</td>
<td>3.5%</td>
<td>6</td>
<td>0.35%</td>
</tr>
<tr>
<td>6</td>
<td>8</td>
<td>2900</td>
<td>2185</td>
<td>715</td>
<td>24.6%</td>
<td>11</td>
<td>0.4%</td>
</tr>
<tr>
<td>6</td>
<td>9</td>
<td>1700</td>
<td>1532</td>
<td>168</td>
<td>9.9%</td>
<td>0</td>
<td>0.0%</td>
</tr>
</tbody>
</table>

4.4.3 Rail gauge measurements

Rail gauge was estimated at run time using a computer installed on the railroad cart. We have tested our system at different track sites in Central Florida. We found that the profiles of the rail gauge often feature a characteristic wave pattern (Fig. 18). We have also observed that the tracks on main railroad lines tend to show far less gauge variation then the tracks which are used more rarely. With our rail gauge measurement technique we were able to discover many defect sites on the rail tracks. We tested the correctness of our rail gauge measurement by running the cart several times over the same track segment. The gauge plots measured in both directions were found to match for all track segments (Fig. 18). We have verified the rail gauge by performing actual gauge measurements on tracks. Since our method is image-based and images have a far superior resolution to other popular data acquisition methods, like lasers, the precision of our gauge measurement method was comparatively very high. In fact, we were unable to determine the precision of our gauge measurement method by comparing it to gauge taken by using a
measuring tape, since it yields a better accuracy than hand measurements. We estimated that the precision was less than 0.1 inch.

Figure 18: The profile of a rail gauge measured in inches at a track segment of about 100 feet long.

### 4.5 Method for camera calibration

For camera calibration we are using a 60 inch long calibration bar. The bar has two sets of 3 vertical black stripes on a light background, which were located at different ends on the bar. The bar is laid perpendicularly on rails so one set of stripes is visible from every camera (Fig. 19a). All stripes are 1 inch wide and the distance between stripes is exactly twice the stripe’s width.

One set of calibration images is taken. We find vertical Sobel edges (Fig. 19b) and then inspect every horizontal scanline of the edge map for the presence of a calibration bar edge signature. We discard edge points which do not belong to vertical edge lines. We also invalidate all edges with distances less than 10 points between them. The sought signature of a calibration bar consists of 6 edges, with the measured distances between edges 1 and 2, 3 and 4, and 5 and 6 being equal, while the distances between edges 2 and 3, 4 and 5 being roughly twice as long. If this signature is found in the three consecutive scanlines or more, the calibration bar is considered found (Fig. 19c).
The image $x$ locations of innermost stripes are set as the camera zero points, and the measured world distance between innermost stripes is $D_{\text{cam}}$. The zoom ratios are automatically found by comparing the stripe widths in the image to the corresponding stripe world widths.

Figure 19: a) The calibration bar as seen from the left camera. The widths of black stripes and the distances between them are known values of 1” and 2” respectively. The distance between the rightmost stripe edge and the corresponding leftmost stripe edge in the right camera is also known. b) The edges generated by the calibration bar c) The calibration bar detection.

4.6 Summary

In this section we have shown how vision methods can be applied to detect parts of railroad tracks based on videos from a pair of unaligned high shutter speed cameras. After railroad track parts were detected, we segmented rails out to find rail edges and calculate the rail gauge. The algorithms were developed in MATLAB and have shown reasonably high speed of several frames per second. The algorithms were tested on long sequences of track imagery from different railroad tracks in Central Florida and always produced consistent results. They are available for evaluation on our laboratory web site.
CHAPTER 5

OTHER VISION METHODS FOR RAILROAD DEFECT DETECTION

5.1 Detecting Rail Fasteners

Rail fasteners (or clips) hold the steel rail to the perpendicular crossties. These fasteners can take several geometries and can differ slightly depending upon their purpose and producer.

The purpose of fastener detection is to locate missing fasteners, or to localize the image areas of bolts which hold fasteners in place. The fasteners and bolts are detected with either MACH filters, SIFT operators, or with the use of SVM with non-linear kernels [Bur98]. Of those three methods of detecting bolts, SVM had shown the least accuracy at approximately 80%, while MACH had shown the highest (see 5.4 and 5.5). Detection of missing bolts and fasteners is important because two misses in a row will likely lead to an accident. A fastener that appears rotated is likely loose and needs to be fixed or replaced.

We will describe the following fasteners, which were encountered during our data collections: steel fasteners, e-clips, and fast clips. Steel fasteners appear to be a rectangular washer held down by a bolt, when viewed from above. E-clips have a shape similar to that of the English letter ’e’, or a horizontal reflection of the letter, and are sometimes known as Pandrol standard fasteners. Fast clips have the shape of the English letter ’w’ and are also manufactured by the Pandrol company. Fig. 20 shows fasteners in our data collections.

Three types of fastening mechanisms that are not part of our study’s focus are Nabla fasteners, Vossloh fasteners, and steel spikes. Our algorithms could be extended to additional classes if
required. Nabla and Vossloh fasteners were never encountered in our data collections, and steel spikes are typically used to fasten rails to wooden sleepers. Our experiments focused on data collections taken on rails lying on concrete sleepers. The exception to this was at rail crossings where an e-clip on a wooden sleeper sometimes appeared.

Other qualities of the data worth mentioning are scale, illumination, and orientation. Since our data was collected from the same cart, with wheels rolling directly on the track, changes in height and scale are small at most. Our platform used active lighting to reduce the variance in illumination in data collection. Lighting variance remained, but its impact was mitigated.

Since all fasteners of the same type look similar, it is convenient to use correlation filters [DHS00, VMJ05] to detect them. Lightning variance is mitigated by using normalizing correlation filters [Lew96], as well as by the cart design. The appearance of the fasteners does not exhibit statistically significant intraclass variation (Fig. 21).
Correlation is neither scale nor orientation invariant. However, it may still be applied to general object detection problems by creating filters at multiple scales and orientations and merging the correlation results from multiple filters over multiple classes. Typically, the speed of correlation approaches is linearly related to the number of filters. Adding more filters can significantly impact the speed of correlation-based detection.

Of correlation approaches, the Optimal Tradeoff Maximum Average Correlation Height (OT-MACH) algorithm was chosen [KCM94, MKS94, ZC99]. This algorithm returns a strong peak relative to the noise provided by clutter. The OT-MACH filter is given by

\[ h = \frac{m_x^*}{\alpha C + \beta D_x + \gamma S_x} \]

where \( \alpha, \beta, \) and \( \gamma \) are nonnegative OT parameters and \( m_x \) is the mean of the vectorized training images in the Fourier domain. \( C \) represents the diagonal spectral density of the additive input noise. Lacking this information, a vector of ones is used instead. \( D_x \) is the diagonal average power spectral density of the training images:

\[ D_x = \frac{1}{N} \sum_{i=1}^{N} X_i X_i^* \]

where \( X_i \) is a vectorized training image. \( S_x \), the similarity matrix, is calculated identically to \( D_x \) with the exception that each input image has a mean image over all training images which is subtracted from it first [MKS94, ZC99]:

\[ S_x = \frac{1}{N} \sum_{i=1}^{N} (X_i - M_x)^* (X_j - M_x) \]

where \( M_x \) is the mean of \( X_i \).
In order to make it more robust to lighting variance, a normalized version of OT-MACH is used. A fast implementation can be found in Lewis [Lew95].

Figure 21: Confusion matrix for the fastener classes using OT-MACH filters.

In order to create the filters, a database of images was collected and ground-truthed. Images of similar classes were grouped and used as training input to produce OT-MACH filters of the same class. In the event that a fastener was cropped in the frames, they were discarded rather than added to the training set. In general, the relative speed of the cart and high-speed camera provided at least one full non-cropped view of each fastener. Fig. 22 shows some detected fasteners of different types.
5.2 Locating Rail Fasteners

The OT-MACH cross-correlation surface peak returns a location, \((x_{im}, y_{im})_t\), for a single image at time \(t\). However, it is useful to localize this image location in the video space \((x_v, y_v)\). By locating the fasteners in video space, the correspondence between the same fastener in multiple frames of video may be made. A simple Euclidean distance threshold in video space is used to determine the correspondence between detections over multiple frames.

Since the data collection is from cameras mounted on the rail cart, only one degree of freedom is present in world coordinates. Likewise, since the cameras are mounted with an image axis parallel to the rail, only one degree of freedom is present in image coordinates. (This assumption ignores the curvature of rail. Even in rail curves, the close-up data appears to be from a straight rail.)

Since it is assumed that consecutive frames have overlapping fields-of-view, SIFT features [Low04] are used to determine the correspondence between two frames. A distance between frames may be approximated by taking the distance between corresponding feature points in the direction parallel to the rail. Specifically, given a consecutive pair of frames, a set of
corresponding feature points is generated [Low04]. Their distances vote for the most likely distance traveled, in pixels, by the camera between the pair of frames.

This relatively simple method has worked without failure. Again, the data is ideal for SIFT feature point correspondence. SIFT works best on highly textured images. Rails are laid over ballast which is typically a pile of rocks and gravel. This background of ballast creates non-uniform and distinguishable texture to identify overlapping regions between two consecutive frames. SIFT’s invariance to translation, scale, and rotation further aid the accuracy of feature point pair correspondence.

Given \((x_{im}, y_{im})_t\), the image coordinates of a correlation peak at time \(t\), it is transformed to the video coordinate system by the following equation:

\[
(x_v, y_v) = (x_{im}, y_{im})_t + (\alpha_x, \alpha_y)_t
\]

Here \((\alpha_x, \alpha_y)_t\) represents the translation in the direction parallel to the rail to shift the first column in frame \(t\) to the first column in frame 1. (Here we assume the lines representing the rail are parallel to the rows in the image.) Its calculation is given next.

\[
(\alpha_x, \alpha_y)_1 = (0, 0)
\]

\[
(\alpha_x, \alpha_y)_t = (\Delta_x, \Delta_y)_t + (\alpha_x, \alpha_y)_{t-1}
\]

for \(t > 1\)

where \((\Delta_x, \Delta_y)_t\) is the dominant translation motion as measured by SIFT feature point correspondence from frame \(t-1\) to \(t\).
Since our data flows horizontally, the $\Delta_y$ value should be small. The $\Delta_x$ should be linear with respect to the speed of the cart on the tracks.

An additional false alarm reduction method is based upon the data given. There were no instances in our data where two fasteners appeared in the same image. In general, sleepers are a regular distance apart although they may appear closer together near rail joints or at road intersections. Even in these instances in our data, there were no instances of two fasteners in the same image. For any given image, only the top correlation peak is considered. For detections in overlapping images which are too far apart to be considered the same object, only the detection with the highest correlation peak is considered while the weaker detection is discarded. Finally, a single global correlation threshold is used over all images to accept or reject a candidate detection.

5.3 Detecting Missing Fasteners

We can use the information about detected fasteners to locate missing fasteners. As trains pass over the rails, they exert an outward force on the rails. The fasteners provide the counter-force keeping the rails in place. Without a strong enough counter-force, the rails can move apart over time, increasing the risk of a derailment where the train falls between the tracks.

Two methods are used to find missing fasteners. The first is direct detection by correlation that applies the identical OT-MACH algorithm. If a fastener is removed, what sometimes remains is the sleeper and the hole that formerly held the bolt that connected the fastener to the sleeper.
Figure 23: a) Here a missing fastener is detected by direct correlation; b) Here a missing fastener is not detected by direct means.

Figure 24: This is a distance-between-fastener plot. Each integer index on the x-axis represents a detected fastener. The y-axis is the distance between a fastener and the previously detected fastener. The fastener that was not detected in Figure 23 can still be found by looking for deviations in the gaps between detected fasteners.

A correlation filter can be made to directly detect sleepers with missing fasteners. Fig. 23a-b shows an example detection. Unfortunately, detection by correlation will fail in many instances. Once a fastener has been removed from the sleeper, the sleeper will often shift. In addition to the
fact that the sleeper may have moved, it is possible an impact has actually damaged the sleeper, completely changing its appearance. This type of problem is unpredictable visually.

However, by accurately detecting the working fasteners, the locations of missing fasteners may be inferred by measuring the average interval and deviation between fasteners in earlier data. Figures 23b and 24 show a missing fastener that was not detected by direct means and the corresponding peak in a distance-between-fasteners plot.

Statistically small gaps between detected fasteners are usually caused by false detections. By predicting the expected location of the fastener, the likelihood of a false detection can be determined independent of the MACH correlation peak value.

Large gaps are usually caused by a missed detection or a missing fastener. By comparing the measured gap to the average gap, the number of missed detections is estimated. Similarly, the location of these missed detections is given and these frames of interest are presented to the field expert for further review.

5.4 Experiments

Out of the evaluation dataset (4.4.1, Table 1), a database of fasteners was created to train correlation filters and measure detection accuracy. Here is a summary table showing the distribution of fasteners in the dataset:

<table>
<thead>
<tr>
<th>Steel fasteners</th>
<th>1905</th>
</tr>
</thead>
<tbody>
<tr>
<td>E-clips (concrete)</td>
<td>475</td>
</tr>
<tr>
<td>E-clips (wooden)</td>
<td>18</td>
</tr>
<tr>
<td>Fast clips</td>
<td>19</td>
</tr>
<tr>
<td>Missing clips</td>
<td>19</td>
</tr>
<tr>
<td>Total fasteners</td>
<td>2436</td>
</tr>
<tr>
<td>Total images</td>
<td>9600</td>
</tr>
</tbody>
</table>
The common steel fasteners and e-clips made up the majority of the data collected. A relative few fast clips, e-clips on wooden sleepers, and missing fasteners were present in the data collected. Most fasteners appeared in two to three frames. Of these, typically only one or two showed the full fastener. The remaining frames clipped the fastener at one of the sides of the image. Approximately two-thirds of all images were in-between fastener locations, thus containing no view of fasteners. These clutter images helped to measure the false alarm rate of the detection algorithm.

Training Dataset. The dataset was randomly divided into 3 groups. Three sets of filters were created. Each filter was created from a combination of two of the three groups within one of five classes. The five classes are steel fasteners, e-clips on concrete sleepers, e-clips on wooden sleepers, fast clips, and missing fasteners. For testing purposes, this allows every fastener in the dataset to be associated with one filter for which the image was not used in training.

Fastener Detection. The OT-MACH algorithm, using these filters, was run over the rail video sequences. Given an image, if a fastener was present, its truth was known and only the filter that did not use this fastener in training was applied to detect the fastener. In all intermediate images that did not contain a fastener, the same set of filters was used.

An ROC curve presenting the probability of detecting fasteners against the probability of false alarms is presented in Figure 25. This probability is out of 2436 possible fastener detections. Included in this are the 19 missing fasteners since an attempt is being made to use a filter to correlate with the typical appearance of a missing fastener.

Approximately 90% of fasteners are detected before the first false alarm. However, some fasteners are not detected without substantially lowering the threshold. The missed detections
Figure 25: Probability of detecting a fastener vs. probability of false alarm. The locations of the 70%, 60%, and 50% thresholds are shown.

Figure 26: a) Left: An example of a "gray" steel fastener. Contrast it with the image to the right. Right: An example of a fastener whose confidence is reduced due to clipping. b) Left: An example of a missing fastener - a defect. Right: An example of an over-saturated image.
can be split into five categories. They are listed based upon their frequency: rotated fasteners, "dark" bolts on steel fasteners, clipped fasteners, missing fasteners, and saturated fasteners.

Some fasteners poorly hold in place and thus appear rotated or shifted. Some other fasteners are not detected due to untypical appearance of fasteners. We discuss how to detect such fasteners in the next section (5.5). The steel fasteners typically appear as a dark gray rectangular plate with a reflective bolt to hold them place. However, there are approximately 100 that have a non-reflective bolt and thus correlate poorly with the trained filter for this class. Other misdetections typically happen due to defects in imagery. The next group of missed detections happens when our data collection cart goes so fast as to give us two images of a fastener: one clipped by the right edge and one clipped by the left edge. These two types are shown in Fig. 26a. The fourth group, missing fasteners, is not surprising, as the visual appearance of missing fasteners is not uniform. The final group is due to a brief segment of our video that gets washed out by light. The images are almost completely saturated. Examples of these last two types are shown in Fig. 26b.

Over the same test dataset for fastener detection, 19 missing fasteners were present; they manifested themselves as spikes in the distance-between-fasteners plot (Fig. 24).

5.5 Detecting Rotated or Untypical Fasteners

Since the distance from cameras to fasteners does not change, the scale of fasteners in the image does not change. However, the angle of view can be different due to two reasons: either the fasteners can be physically rotated, or the cart wheels skid on rail. Since the main drawback of correlation filters is that they are not scale or rotation invariant, we have to apply different filters
to find rotated fasteners. It is common practice to use a different filter if the target is rotated more
than 3 degrees in either direction.

Therefore, if the fastener can be rotated up to 90 degrees, we have to use 30 different filters to
detect it. If any untypical fasteners exist, this also increases the number of required filters. How
can we process so many filters without dropping computation speed below a reasonable level?
The possible answer lies in the use of parallel computation technology which is widely available
in the GPUs of most advanced graphics cards from nVidia and ATI. As of 2008, these cards can
perform parallelizable applications hundreds of time faster than CPUs from Intel and AMD. In
our lab, we have developed a library, MinGPU (see Chapter 6), which facilitates the use of GPUs
for Computer Vision [BS08]. We have also developed a MinGPU-based library of popular
computer vision methods, which includes an implementation of correlation filters on the GPU.
The idea is that, after being loaded into the GPU, a single input correlation filter is used to
generate all the required scaled and rotated correlation filters. Then, the input image is loaded
into the GPU and all those filters are applied to it. Possible scales and rotation angles are hard-
coded in the algorithm. All processing is done directly in the GPU. This makes a rotation and
scale invariant GPU correlation filter which has approximately the same execution time on the
GPU as regular correlation filters on the CPU. We discuss GPU implementation in detail in
Section 6.4.

The detection rate of fasteners with the use of sets of MACH filters on GPU is more than 98%,
with the remaining 2% depending mainly on the quality of the input images.
5.6 Summary

The proposed method was found to be quite successful in detecting different types of railroad elements with fixed or similar appearance, like railroad clips, bolts, and rail plates. The correlation filters, and MACH filters in particular, were found to work the best among other vision algorithms for this purpose. The drawback of the correlation filters has always been that they need to be reapplied if scale or angle of view of the object changes; however, in Section 6.4 we demonstrate that with the use of novel technologies, like the modern GPUs, the application of multiple correlation filters can be done in real-time.
CHAPTER 6
REAL-TIME IMPLEMENTATION

6.1 Introduction. MinGPU library

MinGPU is a library which hides all graphics-related routines from the end user, which is particularly useful for those who do not have particular skills or desire to delve into low-level details of GPU programming. MinGPU can be used as a basic building block for any GPU application. It can help implement any algorithm on GPU. Besides MinGPU, there exists another library which is dedicated to computer vision on GPU. OpenVIDIA [FMA05, OVD] is an open source library which contains many useful vision algorithms, like edge and corner detection, feature-based corner, object tracking, and image enhancement/preprocessing routines. The complete algorithm list can be found at [OVD]. OpenVIDIA also maintains a web page which lists many recent computer vision related papers [CV07]. However, this library is not built upon a reusable core. Another notable effort is the BrookGPU library [BB04]. BrookGPU was conceived as an extension to the C language which supports stream programming and data-parallel constructs. It includes a compiler and run-time support libraries. This library has not been updated since 2004. The latest v0.4 release of BrookGPU can be downloaded from [BG04]. Yet another open source GPU library is Sh [SHGPU]. Sh is a GPU metaprogramming library designed in C++. It defines GPU objects and data types as C++ classes, and also defines operations on these objects in terms of C++. Control structures are defined as C++ macros. Thus,
it is possible to write programs for GPU using C++ notation. However, as the target users for this library are graphics developers, this library serves as a substitute for GPU languages like Cg, GLSL, or HLSL. Both of these medium-sized open-source projects feature little documentation and are complex for users who do not have advanced knowledge of graphics processors and C++ programming. RapidMind [RPM] is a library similar to Sh in that it wraps GPU constructs into C++ classes. In contrast to Sh, the target users of RapidMind are multi-core software developers. RapidMind is an attempt to provide a unified high-level development platform for parallel computations; it works with Cell, multi-core CPUs, and GPUs. This commercial library is also somewhat advanced in use and does not give direct control over GPU processors.

This work makes two contributions to computer vision. First, we have created a C++ class library, MinGPU. This library can function as an introduction to GPU world for researchers who have never used the GPU before. We intentionally designed the library and interfaces to be as straightforward as possible. MinGPU provides simple interfaces, which can be used to load a 2D array into the GPU and perform operations. All GPU and OpenGL related code is encapsulated in the library; therefore users of this library do not need any prior knowledge of GPU. The library is configurable for use in a variety of purposes. It is available online on our web site at [MG08]. Our MinGPU library is designed to work on nVidia 7000 series cards as well as on ATI Radeon cards series R400 (ATI X800) and later versions. We have designed and tested our library on an nVidia 7300LE card, which is a basic card in the 7000 series and is in widespread use. Some portions may be functional on nVidia 6000 series cards as well as on former ATI cards.
In performance evaluations we used, unless stated otherwise, an nVidia GeForce 7300LE graphics card (‘GPU’) installed in a DELL XPS 410 desktop, featuring a Core 2 Duo 2.4MHz processor and 2Gb of memory (‘CPU’).

6.2 Current trends in GPU development

6.2.1 Graphics processors

Most of today’s graphics cards from the largest GPU makers, nVidia and ATI, contain two processors, the vertex processor and the fragment processor.

*Vertex processor*

All graphics cards operate on 3D polygons. Every polygon is defined by (x, y, z) coordinates of its vertices. For example, every triangle is defined by three vertices. The camera vertex is usually set to be at (0, 0, 0). When the camera moves, the coordinates of all polygon points must be recomputed to reflect a change in the camera position. This operation is performed by the vertex processor. Vertex processors are specially designed to perform this operation and therefore are able to optimize the speed of coordinate transformations. After the coordinates are recomputed, the vertex processor determines which polygons are visible from the current viewpoint.

*Fragment processor*

After the vertex processor re-computes all the vertex coordinates, the fragment processor covers the visible portions of the polygons with textures. The fragment processor does this with a help of the ‘shader’ programs. Originally, in the computer graphics world, the purpose of the ‘shader’
program was to add graphic effects to textures like shade (hence comes the name), but now this feature is being inherited by general-purpose GPU users. Up until a few years ago, all shader programs were written in the assembly language. However, as graphics hardware evolved and became capable of executing much larger shader programs a need for a specially designed language became evident. Many contemporary shader programs are C-like programs written in Cg, ‘C for graphics’ language. The Cg language was created by nVidia. nVidia supplies manuals and examples on Cg, which can be found in the Cg Toolkit [CG05].

The most important difference between contemporary CPUs and GPUs is that GPUs run programs concurrently and are SIMD-type processors. The programs are executed for every output texture pixel independently. Therefore, if the GPU has 8 instruction pipelines in the fragment processor, it is able to execute the same program on up to 8 texture pixels simultaneously. Contemporary fragment processors have 4-128 instruction pipelines.

While both the vertex and fragment processors are programmable, we are more interested in the fragment processor because it is specifically designed to work with textures which, in our case, can be viewed as 2D data arrays. Therefore, all algorithms in this paper are designed to work on a fragment processor. On the other hand, a vertex processor is optimized to work with pixels.

At the time of writing, the typical upscale desktop computer is equipped with an Intel Core 2 Duo processor working at 2.4GHz. Let’s roughly compare productivity of this processor to productivity of an nVidia 7300LE (light edition) GPU, which is a commonplace graphics card installed in the same desktop. We assume single-core CPUs, and no SIMD CPU operations are used. The clock rate of 7000 series nVidia GPUs lies in 400-700MHz range. The 7300LE runs at 450MHz. There are 4-24 pipelines in the fragment processor in the currently popular 7000 series.
The 7300LE contains 4 pipelines. We also take into account that nVidia GPU pipelines can typically process two pixels in one clock cycle and they process each of the pixel’s 4 color channels simultaneously. Each pixel is represented as a 4-float number (RGBA). Therefore, if we set up our array so that each RGBA float assumes a data value we gain an additional 4 times the speed. After we multiply all the increases in speed, the nVidia 7300LE works as a processor with a virtual 14.4Ghz rate, due to parallelism; this rate is already 6 times higher than that of the latest Intel Core 2 Duo 2.4GHz CPU. Table 3 illustrates trends in GPU development in recent years.

Consequently, we find that the modest GPU card installed in our computer has a performance about 6 times higher than the latest CPU. If we install the latest nVidia 8800 GPU (1.5GHz, 128 pipelines) we expect to see an additional 106 times increase in hardware performance.

It is hard to make an exact performance comparison of current CPUs to GPUs for many reasons. Modern processors are also equipped with technology that allows simultaneous processing of multiple data by the same instruction. Starting with the Pentium II, all Intel processors are equipped with MMX technology which allows sequential processing of 4 integers. All Intel and AMD processors newer than Pentium III include SSE – similar technology which processes 2 to

<table>
<thead>
<tr>
<th>Graphics Card</th>
<th>Year</th>
<th>Shader Unit Clock rate (MHz)</th>
<th>Shader Units</th>
<th>Texture Fill rate (billion/sec)</th>
<th>Memory bandwidth (GB/s)</th>
<th>Memory max (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>GeForce 8800 Ultra</td>
<td>May 2007</td>
<td>1500</td>
<td>128</td>
<td>39.2</td>
<td>103.7</td>
<td>768</td>
</tr>
<tr>
<td>GeForce 8800 GTX</td>
<td>Nov 2006</td>
<td>1350</td>
<td>128</td>
<td>33.6</td>
<td>86.4</td>
<td>768</td>
</tr>
<tr>
<td>GeForce 7900 GTX</td>
<td>Mar 2006</td>
<td>650</td>
<td>24</td>
<td>15.6</td>
<td>51.2</td>
<td>512</td>
</tr>
<tr>
<td>GeForce 7800 GT</td>
<td>May 2005</td>
<td>550</td>
<td>16</td>
<td>9.6</td>
<td>32.0</td>
<td>256</td>
</tr>
<tr>
<td>GeForce FX 6800</td>
<td>Jun 2004</td>
<td>400</td>
<td>16</td>
<td>6.4</td>
<td>35.2</td>
<td>256</td>
</tr>
<tr>
<td>GeForce FX 5800</td>
<td>Jan 2003</td>
<td>400</td>
<td>4</td>
<td>4.0</td>
<td>12.8</td>
<td>256</td>
</tr>
</tbody>
</table>
4 floating point numbers simultaneously. We also have to mention a recent trend to include more than one processing core on chip. For example, recent Pentium processors feature 2-4 cores which share the same on-chip cache. Every core can be viewed as a separate processor. Nevertheless, the productivity of current graphics processors significantly exceeds the productivity of CPUs, and this trend will continue.

On the other hand, it is not possible to run a transistor-based conventional processor at more than approximately 4-6GHz clock speed due to physical constraints [Mei95]. Theoretically the only way to circumvent this limit lies in the use of different base technologies, like quantum technology. At the time of writing, such technology is many years down the line. Researchers in computer vision are aware that many vision algorithms are not currently able to run in real-time due to their high computational complexity. We feel that the one way to make this possible in the foreseeable future is the use of parallel technology, such as those present in graphics processors.

6.2.2 GPU limitations

All GPUs suffer from two drastic design limitations. The first limitation is, ironically, the fact that GPU computations are done in parallel. The algorithm must be ready to work in multi-thread mode in order to operate on the GPU, which is not a feature of many algorithms. In the GPU, every pixel in the 2D texture is processed independently. It may not be possible to know the sequence in which pixels are processed; therefore, it is not possible to pass any data between pixels while they are being processed. For example, let’s consider a popular connected components algorithm which is used in many vision algorithms today, the Canny edge detector. There exist two versions of a connected components algorithm: recursive and non-recursive. A recursive version cannot be implemented on the parallel processor, because being “recursive”
implies knowledge of the order in which pixels are being processed, which is not present in a parallel processor. A non-recursive version of the connected components algorithm uses a running variable, which contains the largest region label currently assigned. It is not possible, as it was stated above, to maintain this counter on the parallel processor. There exist some parallel versions of connected components [HW90]; however, those versions use binary structures like graphs and trees, which are hard to implement on the GPU. Currently, we do not know of any successful implementation. In the GPU, when one processes a pixel at (x, y) location one can only write the result into (x, y) location of the output texture. There may be more than one output texture (as many as 4-24 for 7000 series and up to 48 in 8000 series). We must also consider that every pixel comprises 4 floats (RGBA value). For the 7300LE card, one can write 16 values for every pixel processed, but they cannot be written into any other than (x, y) location in the output textures. This is the second limitation of graphics hardware. Only the latest CUDA-technology based graphics cards from nVidia allow scatter operations in addition to gather operations.

For example, let’s consider building a color histogram of a grayscale image. Every pixel in the input image may take values in the range of 0 to 255, which means there are 256 bins in the color histogram. For every pixel processed, we must increment one of 256 bins, which due to the above limitation is not possible on the GPU. This is a very simple algorithm, yet it is not possible to implement it on the GPU. One source [OVD] devised an approach which computes approximate color histogram on a small set of input values.

Many other researchers agree that the most promising algorithms to implement on the GPU are filter-like algorithms, which process every pixel independently of the others. Examples of this
include Gaussian smoothing, convolutions, image pyramids, geometric transformations, image
de-noising, and cross-correlation, as well as many other algorithms.

Contemporary computer architecture features one impasse with respect to the GPU which we
cannot avoid mentioning, which is the transfer rate between main (CPU) memory and GPU
memory. The latest computers are equipped with PCI Express memory bus, which is the fastest
expansion bus to date in desktop computers. This memory bus has a full duplex transfer rate of
250MB/s for every lane (500MB/s for PCI Express 2.0 released in January 2007). There may be
up to 32 serial lines, however, many commodity computers are equipped with less than that. We
measured main memory to GPU memory bus transfer rate on our new DELL XPS 410 desktop to
be approximately 860MB/s. At this speed it would take approximately 4-5 ms to transfer an array
of 1 million 32-bit floating point numbers (1k x 1k image) from the CPU to the GPU. Simple
operations (addition, for example) over the same array in the CPU (Core 2 Duo 2.4MHz) would
take about 4 ms. Therefore, the time required to complete simple array operations, such as
transferring an array from CPU to GPU, is comparable to the time required for applying an
operation in the CPU. An example of this backlog is given in the beginning of Section 3; we
would like to point out that some older GPU cards feature slower a GPU to CPU transfer rate
than CPU to GPU. An interesting recent trend in computer design is the appearance of expansion
connectors for HyperTransport buses, the front-side buses used in many of today’s computers as
a fast link between processor and main memory. The HyperTransport protocol supports
41.6GB/s bandwidth in two directions, making it much faster than PCI Express. Recently, plug-
in cards, such as fast speed coprocessors, have appeared and can access the HyperTransport bus.
6.3 MinGPU library

In the area of computer vision, we often encounter situations in which we need to process every point of a 2D array (for example, in an image) with a double loop as demonstrated below:

```c
for (row = 0; row < MaxRow; row++)
{
    for (col = 0; col < MaxCol; col++)
    {
        // do something
    }
}
```

Any 2D array can be represented as a GPU texture. If we do not need to carry over any information between the points of this array, we can implement the inner part of this double loop as a shader (Cg) program. The array is uploaded into the GPU as texture. Then, the shader program is run and the output texture downloaded back into main memory.

In this section we introduce the smallest possible library, MinGPU, which can implement the above mentioned code on a GPU. We attempted to convert the CPU code into GPU in a straightforward manner. In Section 5.3.1, we present an implementation of this double loop in MinGPU. The rest of the section is dedicated to a few more MinGPU examples based on simple vision algorithms.

This section uses a learn-by-example technique. We progress from simple examples, like taking image derivatives and computing image pyramids, to more elaborate examples, such as an implementation of a homography transformation on a GPU. In the following section we present an elaborate GPU example, an implementation of a homography transformation on a GPU.

MinGPU is a C++ class library. Due to the encapsulation paradigm, users do not need any knowledge of its inner structure, therefore they do not need any knowledge of the details of how
the fragment processor or OpenGL drivers operate. MinGPU contains only two classes: Array and Program. The Array Class defines a 2D array in the GPU memory, while the Program class defines a Cg program in the GPU memory. All class methods are listed in Appendix A. In a straightforward scenario, the user prepares the data array and uploads it to the GPU using the methods from the Array class. The Cg program is then loaded and compiled. Program parameters can be set using the method from Program class. The Cg program is then run and the results are generated in the GPU memory, which are then downloaded to CPU memory by another call to a method from the Array class. The ‘Hello, World’ example illustrates this.

6.3.1 ‘Hello, World!’ example
We are going to convert this simple CPU code into GPU:

```cpp
for (row = 0; row < MaxRow; row ++)
{
    for (col = 0; col < MaxCol; col ++)
    {
        Array[row][col] ++;
    }
}
```

The code which implements ‘Hello, World’ on MinGPU is given in Listing 1. This, as well as the most other listings, contains two pieces: a C++ program and a Cg program. Let’s first look at the C++ program. It matches the idea we discussed above in a straightforward way: we create both array and Cg programs, copy the array to a GPU, set program parameters, and run it.

Listing 1: introduction to MinGPU
Array Array;
Program Program;
Array.Create(fpArray, cols, rows, Luminance);
Array.CopyToGPU();
Program.Create(strProgramFile, "main");
Program.SetParameter(enTexture, "texture", (void *) Array.Id());
Program.Run(&Array);
Array.CopyFromGPU();

float4 main (
    float2 coords : TEXCOORD0,
samplerRECT texture) : COLOR
{
    float4 result;
    float4 val = texRECT(texture, coords);
    result = val + 1;
    return result; {or, equally, return 1 + texRECT(texture, coords);}  
}

As for the Cg program, we won’t be able to cover the entire Cg language here (which you can find in a Cg user manual [CGT05]), but we can highlight some key points. First, our original program contains just one function, main, which is its entry point. There are two input parameters to this function, a parameter coords of type float2 and a parameter texture of type samplerRECT. The parameter coords is bound to pre-defined name TEXCOORD0, which contains (x, y) coordinates of the currently processed pixel. Pre-defined names are set automatically by the hardware, so we do not need to provide values for these. Subsequently, the parameter texture is a regular parameter which we must initialize with a call to the SetParameter function. It is our input array, which we want to increment by 1. The standard Cg function texRECT fetches a value of texture array at coords coordinates. In this simplified example, we used the same Cg texture as both an input and an output array. We store the intermediate value in the result variable, a standard 4-float RGBA type. The assignment string result = y + 1 increments each of four float values in result by 1. In this way, every float in result will contain the same value.

Cg functions return one 4-float RGBA value which is the generated value for the current (x, y) texture pixel. When we download this value to the CPU in luminance mode (which is discussed in the following subsection), OpenGL drivers take a mean of the 4 RGBA floats to produce a
single luminance value for each texture pixel. In color mode, all 4 RGBA channels are returned. All Cg programs run on every point in an output, not an input, array. These two arrays can be different sizes.

6.3.2 MinGPU operating modes

The ‘Hello, World’ code has some implicit features which we need to clarify. First, we always assume that MinGPU input arrays contain grayscale values in the 0 to 255 integer range. This array format is common in computer vision. Second, all numbers a GPU operates on are floats of 8-32 bit length. Therefore, MinGPU converts inputted integer arrays into floats. MinGPU uses 32-bit (4-byte) long floats. This is the largest floating point format supported in graphics cards today.

A fragment processor encodes every texture pixel as a four float RGBA value, a quad. There is one float value for the red, green, and blue colors and one for the alpha channel. All operations on a quad are always performed on all four floats in parallel by hardware. The Cg language conveniently includes a special vector type definition float4 to define a quad.

While all MinGPU input arrays are invariably grayscale, MinGPU supports two color modes for arrays: a luminance and a color mode. The reason for this is that GPU color modes are manufacturer-dependent. Since a luminance mode is expected for the nVidia family of processors, it is guaranteed to work in all nVidia cards. However, for all current ATI cards a color mode is required. nVidia also fully supports a color mode. For a list of color modes supported by different cards, see tables in [God08].

In a luminance mode, every float in the quad holds the same value of one input array cell. In a color mode, MinGPU replicates every input value four times, so that each float in a quad
contains the same value. Luminance and color modes are compatible on the level of a C++ code and on the level of a Cg program. In MinGPU, a color mode is specified on a per-texture basis. The luminance mode is the default. Textures can be created in a color mode by setting bMode to enRGBA in a call to Array::Create.

6.3.3 MinGPU Basic examples
In the following listings, we used a reduced notation. For brevity, we implied that all required initialization has already been done and omit array and program initialization code from the C++ program. The example of a simplified ‘Hello, World!’ is given in Listing 2.

Listing 2: reduced ‘Hello, World’
```cpp
... Array.CopyToGPU();
ProgramsetParameter(enTexture, "texture", (void *) Array.Id());
Program.Run(&Array);
Array.CopyFromGPU();
```

6.3.3.1 Taking image derivatives
Taking image derivatives is arguably the most straightforward computer vision algorithm. Image derivatives can be taken in three different directions – \(d_x\), \(d_y\), and \(d_t\). As a derivative kernel, we use Prewitt, Sobel, or Laplacian 3 by 3 gradient operators.

The C++ code and Cg program for taking image derivative in the \(d_x\) direction are given in Listing 3. Texture contains an input array and Kernel contains a smoothing kernel. This code contains one more array, Output, than the ‘Hello World’ example. This array accumulates the derivative results. Initially, the Output array does not contain any values, so we do not copy this array from a CPU to the GPU; we create it right in the GPU instead. The array we use in the call to Output.Create receives the results when we download them from the GPU with the CopyFromGPU method.
Listing 3: image derivative in x, y direction

... Output.Create(NULL, cols, rows, Luminance); Array.CopyToGPU(); Program.SetParameter(enTexture, "T", (void *) Array.Id()); Program.SetParameter(enMatrixf, "K", (void *) Kernel.Id()); Program.Run(&Output); Output.CopyFromGPU();

float4 Derivative3x3 ( float2 C : TEXCOORD0, samplerRECT T, uniform float3x3 K) : COLOR {
    float4 result = 0;
    for (int row = 0; row <= 2; row ++)
    {
        for (int col = 0; col <= 2; col ++)
        {
            result = result + K[row][col] * texRECT(T, C + float2(col - 1, row - 1));
        }
    }
    return result;
}

The programs for completing \( d_y \) derivations are the same, except for the kernel. The Cg program for \( d_t \) derivations must take two arrays, image 1 and image 2, as input so it must be different. The chosen kernels \( K \) for this derivative are a 3 by 3 matrix filled with a value of one and the number one. We included Cg programs for \( d_t \) derivations in Listing 4. Array \( T1 \) is an image at time \( t \) and array \( T2 \) is the same image at time \( t + 1 \).

Listing 4: image derivative in t direction

float4 DerivativeT3x3 ( float2 C : TEXCOORD0, samplerRECT T1, samplerRECT T2, uniform float3x3 K) : COLOR {
    float4 result = 0;
    for (int row = 0; row <= 2; row ++)
    {
        for (int col = 0; col <= 2; col ++)
        {
            float4 p1 = texRECT(T1, C + float2(col - 1, row - 1));
            float4 p2 = texRECT(T2, C + float2(col - 1, row - 1));
        }
    }
}
\[
\text{result} = \text{result} + K[\text{row}][\text{col}] * p2 - K[\text{row}][\text{col}] * p1;
\]

} 

return result;
}

float4 DerivativeT1x1 (  
  float2 C : TEXCOORD0,  
samplerRECT T1, samplerRECT T2) : COLOR
{
  return texRECT(T2, C + float2(col, row)) - texRECT(T1, C + float2(col, row));
}

It must be noted that we cannot use the same texture for both the input and output arrays. This is not possible because the values of the points in the input array are used in calculations of more than one output point. Also, the order of calculations is unknown because all the calculations are done in parallel.

Listing 3 contains the first Cg program with loops. Of all video cards which exist today, only the latest cards from nVidia support loops in hardware; older graphic hardware does not support hardware loops. This is the reason why the Cg compiler unfolds some loops during the compilation. A Cg compiler can only execute trivial loops. Loops with number of iterations dependent on the input parameters cannot be executed. This leads to using a fixed number of iterations in Cg program loops, and consequently multiplies the number of Cg functions. For example, we have to keep a separate \textit{derivateve3x3} Cg function for 3 by 3 derivative kernel, \textit{derivateve5x5} Cg function for 5 by 5 derivative kernel and so on.

6.3.3.2 Computing Image Pyramids
The pyramid computation is another simple vision algorithm [AAB84]. Pyramids are useful data structures for representing image regions. The lowest level of the pyramid is the original image. Each subsequent level in the pyramid is \(\frac{1}{4}\) of the area of the previous level. The process of going from higher to lower resolution is called reduction, while the opposite operation is called
expansion. We have shown only the REDUCE operation, which is computed according to the following formula:

\[ g_l(i, j) = \sum_{m=-2}^{2} \sum_{n=-2}^{2} w(m, n) g_{l-1}(2i + m, 2j + n). \]

In the above equation, \( g_l \) is an image at pyramid level \( l \), matrix \( w \) contains the weight mask, and \( i \), and \( j \) are the indices for the image’s columns and rows, respectively. A C++ code and a Cg program for REDUCE operation are listed in Listing 5.

**Listing 5: image derivative, unfolded loops**

```c++
float4 Derivative3x3 (float2 C : TEXCOORD0, samplerRECT T, uniform float3x3 K) : COLOR {
    float4 result = 0;
    result = result + K[0][0] * texRECT(T, C + float2(-1, -1));
    result = result + K[0][1] * texRECT(T, C + float2(-1, 0));
    ...
    result = result + K[2][1] * texRECT(T, C + float2(1, 0));
    result = result + K[2][2] * texRECT(T, C + float2(1, 1));
    return result;
}
```

In this example, the input and output array sizes do not match. Because each pyramid reduction effectively reduces the image size by half in both dimensions, the output array side is half as long as the input array side. The important question is: how do we determine the values of array elements outside of the array boundaries? In two previous examples, we were able to access such elements. In all of the aforementioned examples, elements located outside of the array are assigned the value of the nearest element in the array. This behavior is hard-coded by a call to OpenGL function `glTexParameteri` during array creation. There are no other options; therefore if we need to assign a predefined value, such as 0, to elements lying outside of the array area, we have to fill our array with 0’s.
We would like to clarify here what we stated in Section 6.2.2 on the limitations of GPU limits. This section may have given the wrong impression that GPU programming is quite simple. In fact, a majority of algorithms either cannot be implemented on current GPUs or can only be implemented with significant difficulties. The problems arise due to two reasons mentioned in Section 6.2.2. First, the fragment processor does computations for all points in parallel and, therefore, the algorithm must be able to work in parallel. In particular, it means that the order in which points are processed is not known. Therefore, there are classes of algorithms which cannot be implemented on a GPU, notably all recursive algorithms. Also, any global scope variables, such as counters, cannot exist in a parallel algorithm, only constants can be used. Another inconvenient and often problematic limitation is that the current Cg programs can write only to the location of the element they are currently processing. Quite often algorithms have to be altered to account for that limitation.

We began this section with the ‘Hello, World’ code, the simplest MinGPU program. Unfortunately, that program does not gain an increase in speed due to the use of the GPU. While execution of the code took 2 ms on our GPU compared to 4 ms on a CPU, there is an additional overhead to transmit the array to and from the GPU, requiring 4 to 7 ms for the 4MB array. Algorithms which operate according to a ‘single array load – single use’ scheme will likely not gain a significant increase in speed by uploading to the ordinary GPU. In fact, if the computational portion is small, it can even result in a decrease in speed. To gain a significant increase in speed from the use of the GPU, an algorithm must operate according to the ‘single array load – multiple use’ scheme. It is also desirable that the computational portion be large.
In the next section, we demonstrate this thesis by presenting an example of an algorithm which operates according to the ‘single load – multiple use’ scheme. We demonstrate that the increase in speed, due to uploading the algorithms built upon the aforementioned scheme to a GPU, may be quite significant. Some algorithms which are sluggish on a CPU can run in real-time on a GPU.

6.4 MinGPU applied to railroad defect detection

In this section, we discuss an implementation of a scaled version of the normalized correlation filters. Many methods of correlation pattern recognition [DHS00, GW92, KMJ05], like OT-MACH filters [MKS94, KCM94], are based upon the normalized cross-correlation (fast implementations of normalized cross-correlation can be found in [BH01, Lew95]). The limitation of the correlation filters is that they cannot deal with template scaling. We try to overcome this limitation by scaling a filter in the pre-defined \([\text{min}, \text{max}]\) range with a step \(S\). This results in a set of filters which are sequentially cross-correlated with an image. The confidence value of a match is obtained by taking the maximum value from the output of all of the filters at the location. The sample application for template scaling is shown on Fig. 27. There, however, may be another case when we need to apply a set rather than one correlation filter to an image: quite often it happens that the matching template can assume slightly different forms (Fig. 28), in which case it is reasonable to use a set of templates instead of one. In this second case, all filters are of the same spatial size, but contain somewhat different templates.
Figure 27: This figure illustrates a possible application for GPU scaled correlation filters. UAV IR night video has occasional inclusions of people of unknown scale which must be detected. The video is pre-processed with thresholds and Harris corner detector, which leaves about 30-150 possible people candidates in every frame. The exact scale of people is unknown because aircraft altitude keeps changing; however, there exists a limiting range for the scales, both upper and lower. People are being detected by applying a set of differently sized correlation filters to every possible position. Resulting detections are then tracked to ensure their consistency.

Figure 28: The correlation filters implemented on GPU. Railroad track video is made with a camera fixed over tracks, so the scale of objects in a field of view does not change. Our purpose is to detect and count rectangular rail fastener clips which hold the rail in place. However, besides being spatially rotated for up to 30 degrees, clips can be of slightly different geometric/visual appearances by manufacturer’s design. In a set of filters we put a filter for every possible filter appearance. Filters are created using OT-MACH algorithm with a rotation step of 3 degrees, and then applied sequentially to an image.

Formally, the normalized cross-correlation $\gamma(u,v)$ between an image and a filter at every point $(u,v)$ in an image is computed according to the formula [Lew95]:

$$
\gamma(u,v) = \frac{\sum_{x,y} (f(x,y) - \bar{f}_{u,v})(t(x-u, y-v) - \bar{t})}{\sqrt{\sum_{x,y} (f(x,y) - \bar{f}_{u,v})^2 \sum_{x,y} (t(x-u, y-v) - \bar{t})^2}}.
$$
In this formula, \( f(x, y) \) is an input image of \( M_x \times M_y \) size, \( \bar{f}_{u,v} \) is an average of the input image over a filter area, \( t \) is a square filter of \( N \times N \) size, and \( \bar{t} \) is a mean of a filter. The sums \( x, y \) are computed under the window containing a filter \( t \).

6.4.1 MinGPU implementation

Obviously, it is wise to compute this formula in steps by accumulating and re-using intermediate values. First, we pre-compute all of the sums, then we compute the numerator, denominator, and the final value. This computation is repeated for every filter. We maintain a set of filters of different sizes and apply them to every input image sequentially. The highest absolute value for a filter response is then returned as the result.

Calculation of the above formula takes 8 steps. Here are the steps for the same-sized filters:

1. Calculate \( \bar{f}_{u,v} \) for every image point.

2. Calculate normalized value of \( f \): \( f(x, y) - \bar{f}_{u,v} \) for every image point.

3. Calculate \( \sum_{x,y} (f(x, y) - \bar{f}_{u,v})^2 \) for every image point.

4. In a loop for every filter in set:
   i. Calculate a mean of each filter \( \bar{t} \).
   ii. Calculate normalized \( t \): \( t(x-u, y-v) - \bar{t} \) for every filter \( t \) point.
   iii. Calculate a value of \( \sum_{x,y} (t(x-u, y-v) - \bar{t})^2 \).
   iv. Calculate the whole formula, a numerator and a denominator.
   v. In an accumulator array, accumulate a maximum filter response value \( \gamma(u, v) \) for every point \( (u, v) \) in an image \( f \).
For differently sized filters all steps are the same, but steps 1-3 are also included under loop 4, because sums in steps 1-3 are calculated over a filter area which is variable in case of differently sized filters.

The C++ program which implements all steps above is not given in Appendix B; we omit it for brevity. Listing 6 shows Cg programs which calculate averages over an input image (steps 1, 3, omitting trivial subtraction in step 2). Averages and sums over the filter area are similar to averages over the image, so Cg programs for those averages are also omitted here. Listing 7 shows a Cg program which calculates the main formula (step 4iv). All other listings can be found in a source code [MG08].

**Listing 6: normalized cross-correlation**

```c
class Listing 6: normalized cross-correlation

{ computing the image average under the 77 by 77 shifting window }
float4 favg ( 
    float2 C : TEXCOORD0,
    samplerRECT I) : COLOR
{ 
    float4 result = 0;
    for (int row = 0; row < 77; row ++)
    { 
        for (int col = 0; col < 77; col ++)
        { 
            result = result + texRECT(I, float2(C.x + col, C.y + row));
        }
    }
    return result / (77.0 * 77.0);
}

{ computing the squared sum under the 77 by 77 shifting window }
float4 Snorm2 ( 
    float2 C : TEXCOORD0,
    samplerRECT fnorm) : COLOR
{ 
    float4 val = 0;
    float4 result = 0;
    for (int row = 0; row < 77; row ++)
    { 
        for (int col = 0; col < 77; col ++)
        { 
            val = texRECT(fnorm, float2(C.x + col, C.y + row));
        }
    }
    return result / (77.0 * 77.0);
}
```
result = result + val * val;
}
return result;
}

Listing 7: main formula in normalized cross-correlation
float4 Correlation (float2 C : TEXCOORD0,
samplerRECT fnorm, samplerRECT tnorm,
samplerRECT Snorm2, samplerRECT Tnorm2) : COLOR
{
  float4 numerator = 0;
  float4 denominator = sqrt(texRECT(Snorm2, C) * texRECT(Tnorm2, float2(0, 0)));
  for (int row = 0; row < 77; row++)
  {
    for (int col = 0; col < 77; col++)
    {
      numerator = numerator + texRECT(fnorm, float2(C.x + col, C.y + row)) * texRECT(tnorm, float2(col, row));
    }
  }
  return abs(numerator / denominator);
}

Double for loops in Listing 6 are of some interest because of hidden overflow potential they may generate. According to Cg specification, basic integer type int is a 32-bit type. However, for loops in Listing 6 do not accept any values over 127 for integer counters, thus prohibiting use of filters larger than 127 by 127. This peculiar feature has an unknown origin and possibly pertains to 8-bit limitation of assembly format of the for instruction. It is possible to use floats in place of integers as loop counters in Cg. In this case, there is no 128 (byte) limit, but yet another problem arises for program 1 in Listing 6: filter sizes of more than 110 fail if we use float loop counters. No error messages are generated in both cases, so it is not possible to identify the certain cause of errors; however the second problem likely lies in the limited length of Cg assembly programs. Cg programs in all contemporary GPUs, including series 8000, are limited to 65,535 assembly instructions, either dynamic or static. While the static size of the assembly program 1 in Listing 6
is tiny, its dynamic size (the number of executed instructions) may exceed this limit. In both overflow cases, the calculations went completely wrong with no error messages generated.

Because a Cg loop can not be based on program parameters, all filter sizes must be hard-coded in both C++ and Cg programs. The size bounds of filters $L$, $H$, as well as a size step $S$ are thus fixed for every executable module. For a C++ program, a possible coding solution is to use a C macro, while Cg does not support macro or name definitions.

6.4.2 Time considerations

In this section we compare speed of this GPU algorithm to speed of the built-in $\text{normxcorr2}$ function in Matlab (source file $\text{mach.h}$) on the same data set. Both methods produce the same results which also verify the correctness of our approach.

While this is obviously a ‘single load – multiple use’ scheme algorithm, its GPU implementation has its hidden drawbacks. If we look at the eight steps above, we see that steps 1, 3, 4i, and 4iii can be very efficiently computed recursively [BH01]. Instead of calculating multiplications in the entire $N \times N$ filter area, it is ample enough to make $N \times 1$ subtractions and $N \times 1$ additions when we move to the neighboring point in the image. This effectively lowers time complexity for steps 1, 3, and 4 from $O(M_x \times M_y \times N^2)$ to $O(M_x \times M_y \times N)$. However, because the GPU does not support recursive programs, we have to perform the entire set of multiplications on the GPU. Thus, measured time difference between the GPU and the CPU is not only the function of difference in their performances, but also the inverted function of $N$, the filter size. The bigger the filter, the more advantage the CPU will have over the GPU.

In our tests, we used an image of size $160 \times 120$ and 7 filters of size $77 \times 77$. For both the GPU algorithm and the CPU’s Matlab $\text{normxcorr2}$ function, we measured times needed to perform
actual computations, not including disk and screen input/output operations and auxiliary computations but including memory bus input/output for the GPU. The GPU finished one filter in 0.620s and 7 filters in 1.660s, while \textit{normxcorr2} finished 7 filters in 0.575s. This gives a good example that even in a ‘single load – multiple use’ scheme, the GPU does not necessarily give an advantage over the CPU - everything depends on the particular algorithm. On the other hand, if we had installed the latest series 8800 video card, which has over 26 times the performance compared to our series 7300 video card, then the GPU would have had about 9 times advantage over the CPU. Also, if filter size is decreased the GPU quickly becomes on the par with the CPU. Thus, the algorithm must be particularly suitable for the GPU in order to get its performance improved greatly. The next section gives an example of such an algorithm.

6.5 Discussion

We should note here that neither nVidia nor AMD disclose details about the inner structure of their processors. So, there is some inherent bias in C++ CPU to Cg GPU time comparisons. However, we can make some hypotheses about the reason why this particular algorithm worked much faster than expected.

In today’s computers, the onboard installed memory (DRAM) tends to be hundreds of times slower than memory installed on both CPU caches. Therefore program execution time heavily depends on the cache hit rate, which is typically about 90-98% for modern computers. If we make a rough estimate of a marginal case when GPU cache hit rate for both reads and writes is equal to 100%, we find that the GPU with 100% cache hit rate will work up to a hundred times faster than the CPU with 90% cache hit rate.
There are two reasons to believe that a GPU’s cache hit rate is higher than the corresponding CPU’s. We already stated that when the fragment processor processes texture, it cannot write into a position other than the position of a current pixel. Therefore for current pixel, \((x, y)\), the shader program can write an output only to a position \((x, y)\) in the output texture. This is a limitation of current GPUs. However, there is a flip side to this limitation – it is excellent for the cache write optimization, because the memory write address is always known. Therefore, it is possible to attain a 100% cache write hit rate for the GPU. Second, because the same program is run for every pixel in the texture, it often results in a predictable pattern of memory reads. Unless you are using the conditional statements in your GPU program, memory reads have a predictable pattern. Therefore, it is natural to expect that the cache read hit rate will be higher for the GPUs than for CPUs. The papers by Govindaraju et al. [GM07, GLG06] give some further insight into cache-effective memory models for scientific processing on a GPU.

Table 4: Comparative execution times for homography transformations algorithm.

<table>
<thead>
<tr>
<th></th>
<th>Time Per Slice</th>
<th>Total Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>MATLAB (for loops)</td>
<td>10.5 min</td>
<td>Hours</td>
</tr>
<tr>
<td>MATLAB (built-in functions)</td>
<td>2 min 35 s</td>
<td>Hours</td>
</tr>
<tr>
<td>C++ CPU</td>
<td>12 s</td>
<td>about 25 min</td>
</tr>
<tr>
<td>GPU</td>
<td>0.02 s</td>
<td>3.5 s (including 1.5s file read)</td>
</tr>
</tbody>
</table>

We would like also to mention here an observed speed improvement of 7,750 times over similar Matlab code. This means that while Matlab takes about three hours to compute a single transformation, MinGPU does the same in less than 2 seconds! The time to load input images
from a hard drive was not included; it is usually an additional 1-5 seconds, depending on the hard drive model, operating system state and other parameters.

If we increase or decrease the number of views or slices, the execution time increases or decreases likewise; this means that the execution time is dependent on the number of views and slices. However, tests have shown that the execution time increases exponentially if the image size exceeds a certain threshold. This threshold seems to vary depending on the video card used; therefore it is a hardware threshold. For our nVidia 7300LE graphics card, we found it at approximately 6MB (Fig. 29). This threshold roughly corresponds to the size of the installed GPU memory cache. If the total amount of data accessed by the Cg program exceeds this threshold, processing time grows according to exponential law.

![Figure 29: Time versus image size dependence for algorithms run on the GPU.](image)

We have also acquired the latest nVidia GeForce 8800 Ultra video card and performed our experiments with the homography transformation algorithm on the same input data set. The GeForce 8800 video card was installed into the same desktop computer. We have found that
homographies GPU code completes in about 0.35s on the 8800 card, which is less than the speed increase we predicted in Section 2. We have to note that there are many factors which contribute to the speed, such as a time required for executing C++ code on the CPU, time to initialize graphics libraries, time slice taken by the operating system, and others. We have discovered that C++ code compiled in debug rather than release mode takes 3-5 times more time to execute. So, such speed comparisons are subjective.

We would like to mention here an interesting feature coming with those new cards: the CUDA technology [NVCU]. CUDA works only with the latest 8800 series nVidia cards [NVGF] and Tesla, which makes it a blend of software and hardware technology. CUDA allows us to define shader programs (‘kernels’) in terms of the C code rather than in terms of shader languages like Cg, GLSL, or Sh. The CUDA toolkit contains a pre-compiler which compiles such C kernels directly into the device binary code. This code can then be linked to a C++ compilation by a host compiler like Visual Studio or sent to an 8800 video driver. Thus, users who have access to 8800 cards are not constrained to a use of shader languages. However, these cards are now the most expensive graphics cards on the market, the costs of which routinely exceed the costs of a desktop computer. This makes CUDA unavailable for many applications.

The other interesting features of 8800 series graphics cards are that some hardware limitations are lifted. The cards now support data scatter operations in addition to gather operations. Also, the neighboring threads are assembled into warps (currently collections of 32 threads), which can use the shared memory and synchronization services. While neither CUDA nor 8800 cards are typically capable of performing the entire computation on the GPU, and this technology does not
prevent the user from knowing the intrinsic details of the GPUs, those new developments are promising and allow us to look optimistically into the future.
CHAPTER 7
SUMMARY AND FUTURE WORK

In this dissertation, we have shown that Computer Vision methods can be successfully used to build effective applications for the monitoring of various defects of rail track, applications that are capable of real-time processing of large volumes of field data and reliable detection of railroad defects. We have discussed the advantages and disadvantages of such applications. In the future, this work can be used to build a complex system that monitors the safety of railroad operations. The system and Computer Vision approaches we developed can be used for future hardware design.

7.1 Summary of Contributions

- The computer vision method for rail gauge calculation from the pair of laser range scanners, including
  (a) The method for rail detection from the noisy depth images
  (b) The method for laser’s calibration, including the method for alignment of two lasers
  (c) The real-time implementation and verification on the field data

- The computer vision method for rail gauge calculation from the pair of synchronized high-speed shutter cameras, including
  (a) The method for rail detection in images
  (b) The methods for cameras’ synchronization and calibration
(c) The real-time implementation and verification on the field data

- Other methods for railroad defect detection
  
  (a) The method for detecting faulty or misplaced railroad elements, including
      
      (1) rail fasteners,
      (2) tie plates,
      (3) tie clips,
      (4) e-clips,
  
  (b) The real-time application of the above methods (see next contribution)

- MinGPU – the parallel library of computer vision methods implemented on GPUs
  
  (a) correlation and MACH filters
  
  (b) Optical Flow (Lucas-Kanade)

  (c) fast image transformations and homographies (600 times faster than C CPU code)

  (d) edge detectors (Sobel, Canny) and discrete filters

  (e) image pyramids, morphology operations, and some graphics primitives
7.2 Future Work

The algorithms we developed in this thesis can be extended or improved in a number of different ways. We give some ideas in the following passage.

The list of detected track defects can be much extended and new algorithms can be added. For example, there is a possibility of an algorithm that detects incorrectly placed crossties. Two or three adjacent incorrectly placed crossties indicate a potential problem place on a track. The conditions of rail joints and how to evaluate them can be a topic of a separate work. The levels of ballast and the vegetation can also be measured. There can be potentially defective places directly under the rails where the ballast is missing; they can be uncovered with other algorithms. The cracks and rail burns on the surface of the rail can be a matter of separate investigation. It is possible to develop a vision-based system which will monitor the conditions on a railway station, count the numbers of wagons, and measure the load/use of the different station parts. It is possible to develop a vision-based verification system that will check whether the train operator passed on the red light as well as other dangerous conditions such as a possibility of train collision or problems (‘sun kinks’, or others) with the track ahead.

With respect to the algorithms developed in this work, the algorithm for rail gauge measurement using the laser approach described in Section 3 would benefit from the transition of its parts to the GPU, which will increase the number of scans we process per second. The algorithm for rail gauge measurement using the camera approach described in Section 4 can be translated from the Matlab environment to C. Yet another separate work can be dedicated to rail gauge measurement algorithms using the laser triangulation approach; we have investigated this type of algorithm.
and discovered that due to hardware/physics reasons they are not suitable for outside use in direct sunlight. However, this type of hardware can still be used for underground/metro installations. They will need the appropriate algorithms, which we only briefly studied. The other types of potentially useful and interesting hardware not covered in this work are stereo cameras and line-scan cameras. They all have their benefits and drawbacks and can be optimal/non-optimal to solve a particular problem.

The success and the positive rates of defect detection depend, in part, on quality industrial sensors being available to researchers. For example, the best available laser sensors we used for the experiments in Section 3 have a measurement error in the range of +/- 0.5 inch (Section 1.3) which is comparable to the required precision of an experiment, which created many additional difficulties. However, this segment of hardware is quickly developing today and we can expect the sensor quality to significantly improve in coming years. This will open the way to newer and better algorithms for railroad defect detection. Better and faster computers will allow doing more algorithm iterations per second while increasing the output precision.

All experiments we have done were carried out on the railroad tracks in Orange County, in or around the city of Orlando. This limited the number and types of railroad tracks available for experiments. More refined analysis can be performed if we acquire data from more remote locations, including locations out of state.
LIST OF REFERENCES


[GPU05] Introduction to Image Processing on GPU. nVidia Technical brief, 2005.


[NVGX] NVIDIA(R) GeForce 7950 GX2 Technical Specifications.


Fastening Elements for Railroad Maintenance. IEEE-ITSC International Conference on Intelligent


[Stu96] Stubberud, P.A.: 2-D linear phase frequency sampling filters and 2-D linear phase frequency sampling


[ZC99] Zhou, H., Chao, T.H.: MACH Filter Synthesizing for Detecting Targets in Cluttered Environments for Gray-


[MG08] MinGPU source: www.cs.ucf.edu/~vision\MinGPU.

[MGG08] MinGPU Google group: http://groups.google.com/group/MinGPU.

