2009

Design and implementation of a real-time miniaturized embedded stereo-vision system

Siddhant Ahuja
University of Windsor

DESIGN AND IMPLEMENTATION OF A REAL-TIME MINIATURIZED EMBEDDED STEREO-VISION SYSTEM

by

SIDDHANT AHUJA

A Thesis

Submitted to the Faculty of Graduate Studies

through Electrical and Computer Engineering

in Partial Fulfillment of the Requirements for

the Degree of Master of Applied Science at the

University of Windsor

Windsor, Ontario, Canada

2009

Your file Votre référence
ISBN: 978-0-494-57594-9
Our file Notre référence
ISBN: 978-0-494-57594-9

Declaration of Co-Authorship/Previous Publication

I hereby declare that this thesis incorporates material that is result of joint research, as follows:

This thesis also incorporates the outcome of a joint research undertaken in collaboration with Mr. Bahador Khaleghi under the supervision of Professor Jonathan Wu. The collaboration is covered in Chapter 3 of the thesis as the design and development of the software algorithms and in Chapter 4 as the quantitative comparison of the performance of rank vs. census transform. In all rest of the cases, the key ideas, primary contributions, experimental designs, data analysis and interpretation, were performed by the author.

I am aware of the University of Windsor Senate Policy on Authorship and I certify that I have properly acknowledged the contribution of other researchers to my thesis, and have obtained written permission from each of the co-author(s) to include the above material(s) in my thesis.

I certify that, with the above qualification, this thesis, and the research to which it refers, is the product of my own work.

This thesis borrows some of its material from two original papers that have been previously published, as follows:

1. S. Ahuja, B. Khaleghi, and Q. M. J. Wu, "A New Miniaturized Embedded Stereo-Vision System (MESVS-I)," in *Canadian Conference on Computer and Robot Vision, 2008 (CRV '08)*, 28-30 May 2008, pp. 26-33.

2. S. Ahuja, B. Khaleghi, and Q. M. J. Wu, "An improved real-time miniaturized embedded stereo vision system (MESVS-II)," in *IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops, 2008 (CVPRW'08)*, Anchorage, Alaska, 23-28 June 2008, pp. 1-8.

I certify that I have obtained a written permission from the copyright owner(s) to include the above published material(s) in my thesis. I certify that the above material describes work completed during my registration as graduate student at the University of Windsor. I declare that, to the best of my knowledge, my thesis does not infringe upon anyone's copyright nor violate any proprietary rights and that any ideas,

techniques, quotations, or any other material from the work of other people included in my thesis, published or otherwise, are fully acknowledged in accordance with the standard referencing practices.

Furthermore, to the extent that I have included copyrighted material that surpasses the bounds of fair dealing within the meaning of the Canada Copyright Act, I certify that I have obtained a written permission from the copyright owner(s) to include such material(s) in my thesis. I declare that this is a true copy of my thesis, including any final revisions, as approved by my thesis committee and the Graduate Studies office, and that this thesis has not been submitted for a higher degree to any other University of Institution.

Abstract

The main motivation of the thesis is to develop a fully integrated, modular, small baseline (<=3cm), low cost (<=CAD$600), real-time miniaturized embedded stereo-vision system which fits within 5x5cm and consumes very low power (700mA@3.3V). The system consists of two small profile cameras and a dual-core embedded media processor, running at 600MHz per core. The stereo-matching engine performs sub-sampling, rectification, pre-processing using census transform, correlation-based Sum of Hamming Distance matching using three levels of recursion, LRC check and post-processing. The novel post processing algorithm removes outliers due to low-texture regions and depth-discontinuities. A quantitative performance of the post processing algorithm is presented which shows that for all regions, it has an average percentage improvement of 13.61% (based on 2006 Middlebury dataset). To further enhance the performance of the system, optimization steps are employed to achieve a speed of around 10fps for disparity maps in MESVS-I and 20fps in MESVS-II system.

Acknowledgements

First and foremost, I would like to thank my thesis advisors Dr. Jonathan Wu, for his invaluable guidance and support throughout this research work. I also thank Dr. Narayan Kar, my departmental reader, as well as Dr. Dan Wu, my external reader, for their own advice toward the completion of my research and this thesis.

My thanks also go out to various other faculty and staff at the University of Windsor who in one way or another supported my work. The department technicians, Mr. Don Tersigni and Mr. Frank Cicchello, and the Technical Support Centre staff provided materials and expertise instrumental to the experiments herein.

Finally, I would like to express my gratitude to my colleague Mr. Bahador Khaleghi as without his support and collaboration, it would be impossible to realize this research work.

The work is supported in part by the CRC program, the NSERC Discovery Grant, and the AUTO21 NCE.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

The process of obtaining geometric models in three-dimensions that represent with precision a real world object is referred to as shape/range/depth acquisition. It finds applications in several areas, including photogrammetry, archaeology, reverse engineering, robotic guidance, virtual reality, medicine, cinema, game programming, and others. A precise 3-D model is hard to acquire and numerous methods can be found in the literature which are used to acquire shapes of objects, each with their own advantages and disadvantages. Figure 1 shows the taxonomy of shape acquisition techniques.

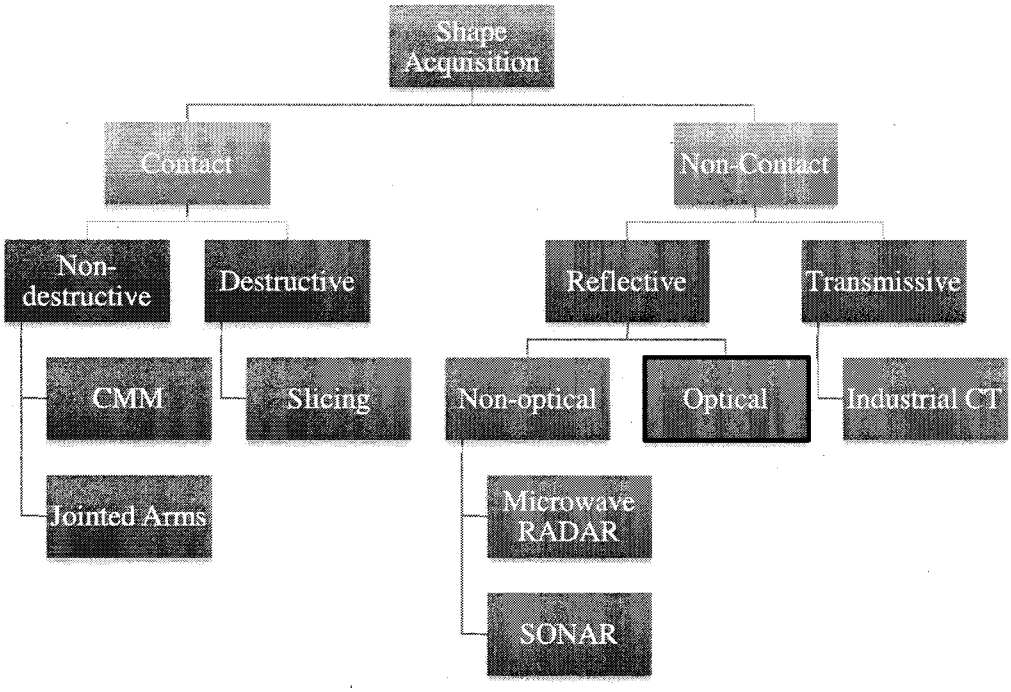


Figure 1. Taxonomy of shape acquisition techniques.

Contact measurement techniques produce highly accurate 3D shapes; however the equipment used is often very expensive and requires the presence of an operator. On the other hand, optical range scanning methods are less invasive (non-contact), safer to use, less expensive, and relatively faster than

non-optical techniques. However, they suffer from many disadvantages including high sensitivity to transparencies, specularities, occlusion, depth-discontinuities, inter-reflections and low texture regions. Figure 2 shows the taxonomy of optical depth measurement techniques.

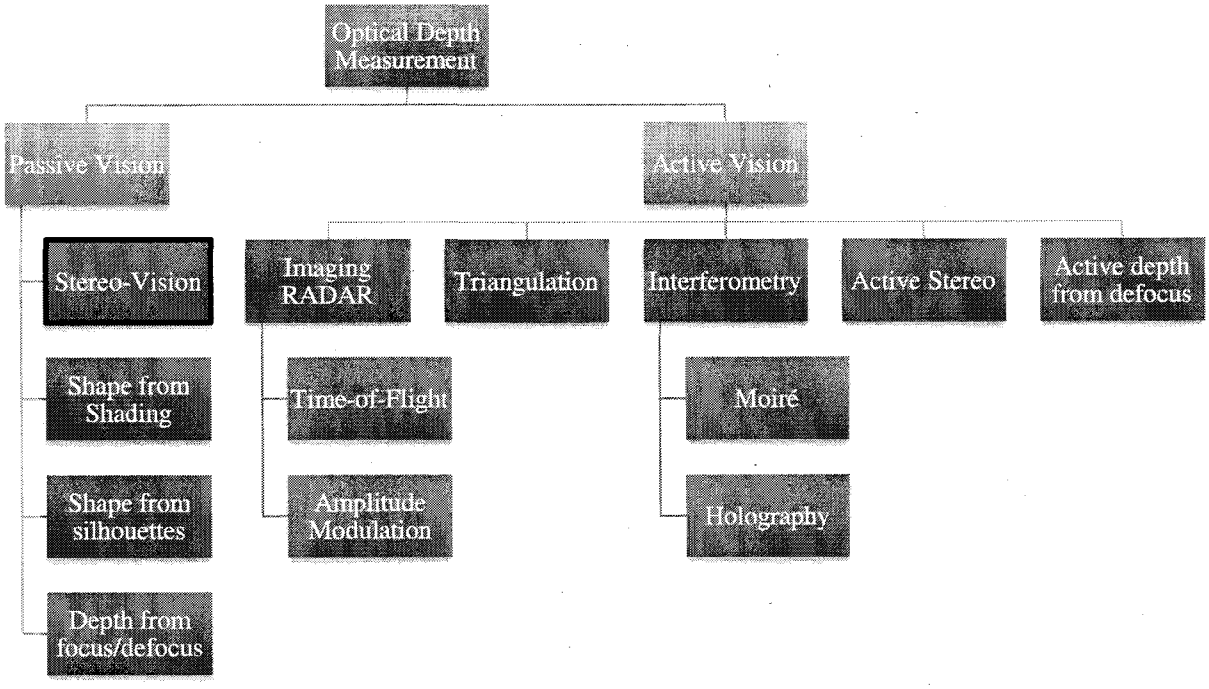


Figure 2. Taxonomy of optical depth measurement techniques.

Active vision techniques require the application of external power for the operation of the sensors. The excitation signal is modified by the sensor to produce an output. Even though they are highly efficient (often matched to the target characteristics), and enable long range operations, they are restricted to the frequencies that can be generated and radiated easily. This excludes part of the far IR (Infra-red), the UV (Ultra-Violet) and gamma ray spectra. They are more complex and less reliable as compared to the passive vision techniques.

Passive vision techniques rely on sensors that directly generate an electric signal in response to a stimulus. They do not emit radiation, and thus enable covert operations. However, they rely on a locally generated or natural source of radiation (sunlight) or a field (gravity), and are prone to feature ambiguity and errors of scale. Typically, they can operate from ELF-Extremely Low Frequencies (<3x103 Hz) to

gamma rays (>3x1019 Hz). Even though they offer good reliability due to the inherent simplicity, the availability at all times is not guaranteed (contrast, light levels etc.).

Shape from shading is one of the forms of passive vision techniques which estimates surface normals and integrates to find the surfaces. It assumes that the surface has a known, constant reflectance, and is surrounded by known lighting condition. It employs a similar mechanism to human vision system, can work with a single image, and does not require correspondence estimation. However, it suffers from the problem of ambiguity, is mathematically unstable and is not very practical.

Shape from silhouettes involves detecting 2D set of closed contours that outline the projection of the object onto the image plane. It requires taking a lot of images from different positions around the object, after which the silhouette is segmented. The larger the number of images acquired, the better is the fit of the final 3D shape. However, due to occluded regions, and non-convex object shapes, this technique only results in a crude model of the real world.

Shape from focusing and defocusing relies on capturing multiple frames by stepping the focal point of the camera in order to estimate the structure of the 3D object. Passive version of this technique is very impractical for capturing shapes of real-world objects.

Stereo vision technique gives depth, which is typically lost in perspective projection. Stereo vision systems take advantage of the fact that the depth of the objects in the scene can be inferred from the relative displacements, also called disparities, of the objects in the scene, when observed from two viewpoints, separated by a distance. Figure 3 illustrates the way we perceive depth.

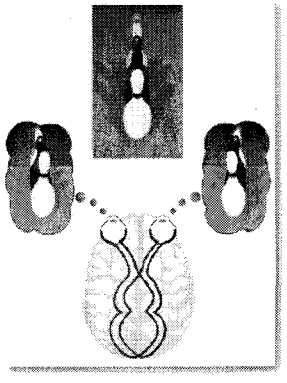


Figure 3. Human visual depth perception [1].

The main objectives of the stereo-vision technique are to find:

a. Correspondence geometry (given a point x in the first view, what is the position of the corresponding point x' in the second image?),
b. Camera geometry (given a set of corresponding image points, what is the geometric transformation between the views?), and
c. Scene geometry (given corresponding image points and geometric transformation between the two camera views, what is the position of the point x in 3D space?).

Figure 4 shows the epipolar geometry or the canonical configuration of the two cameras with parallel optical axis.

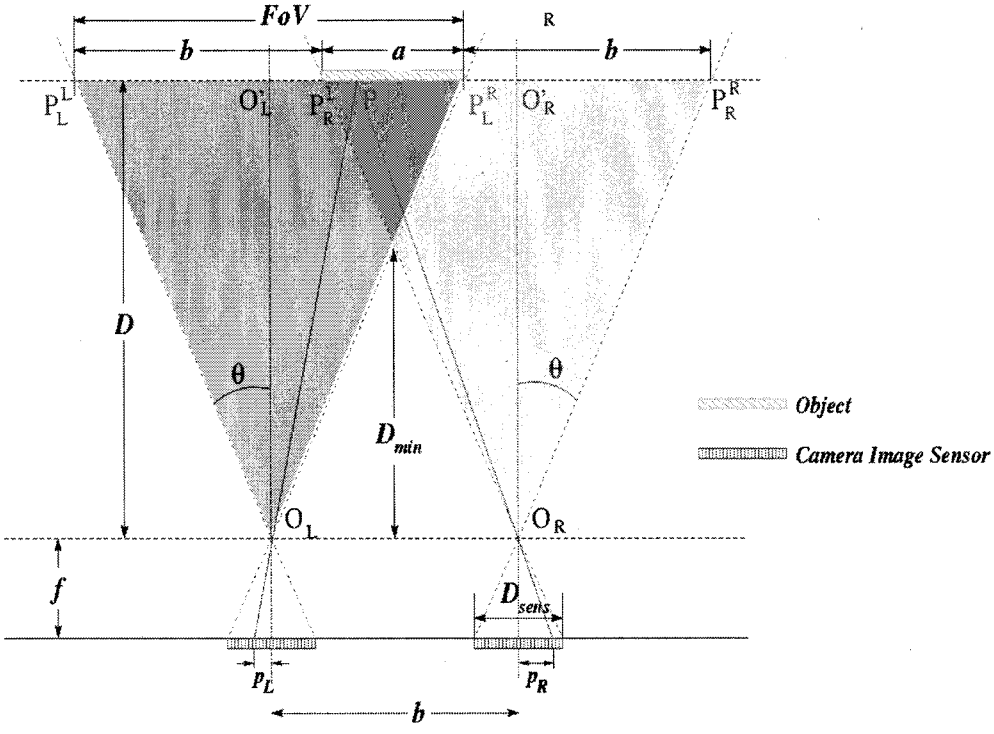


Figure 4. Epipolar geometry-Canonical configuration of two cameras with parallel optical axis.

Epipolar geometry is the projective geometry between two optical views which are independent of the scene's structure, and only depend on the cameras' internal and external parameters. Based on these

parameters, search can be restricted to only one dimension across the epipolar line. The final depth can be calculated from the following equation:

$$Depth = \frac{b * f}{Disparity} \tag{1}$$

where,

b is the baseline (camera separation), and f is the focal length.

In Figure 4,

q is the camera angular FoV (Field of View),

D_{sens} is the sensor width,

n is the number of pixels,

p is the pixel width ,

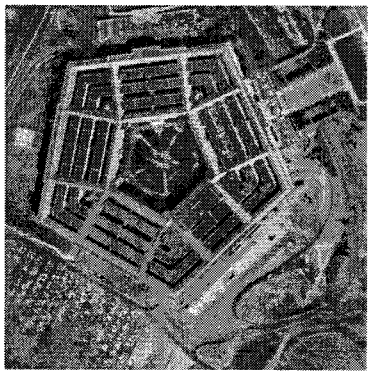
a is the object extent,

D is the distance to object,

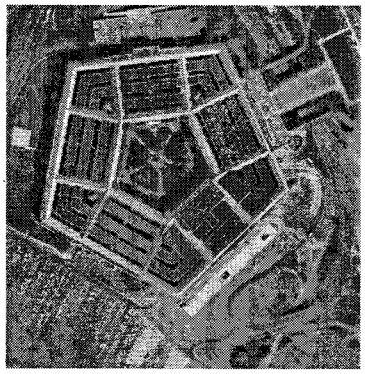
O_L is the left camera's optical centre, and

O_R is the right camera's optical centre

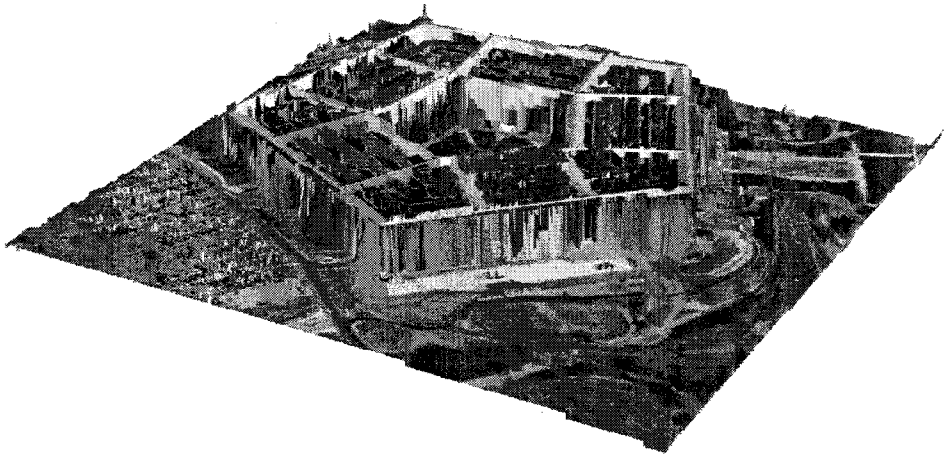
Some of the applications of the stereo-vision technique include 3D scene reconstruction, miniaturized mobile robotics, 3D object tracking, industrial automation, random bin-picking, volume measurement, automotive part measurement, topographical survey, and view synthesis amongst others. Figure 5 shows the left and right images from the pentagon dataset and the corresponding 3D reconstructed view based on correspondence between the images.



Left Image



Right Image



3-D Reconstructed view based on correspondence

Figure 5. Left and right images from the pentagon dataset (top), and the corresponding 3-D reconstructed view based on correspondence between the images (bottom)[1].

There are two main categories of methods used in the field of stereo vision to compute the final disparity map-local (also known as area-based method) and global methods [2]. In local (window-based) methods, the disparity computation for a given pixel is dependent on the intensity values of the pixels surrounding it. By aggregating the support within a window, there is an implicit assumption made of smoothness. On the other hand, global algorithms tend to minimize the energy or cost function by making an explicit smoothness assumption and optimizing the disparity computation problem. Global algorithms tend to be computationally intensive and are not suitable for fast hardware implementations,

[1] Geiger, Davi, Lecture 6, http://www.cs.nyu.edu/courses/spring08/G22.2271-001/index.html

even though they produce a high quality disparity map, as compared to the local algorithms. Figure 6 shows the common local, global and hybrid approaches to finding correspondence.

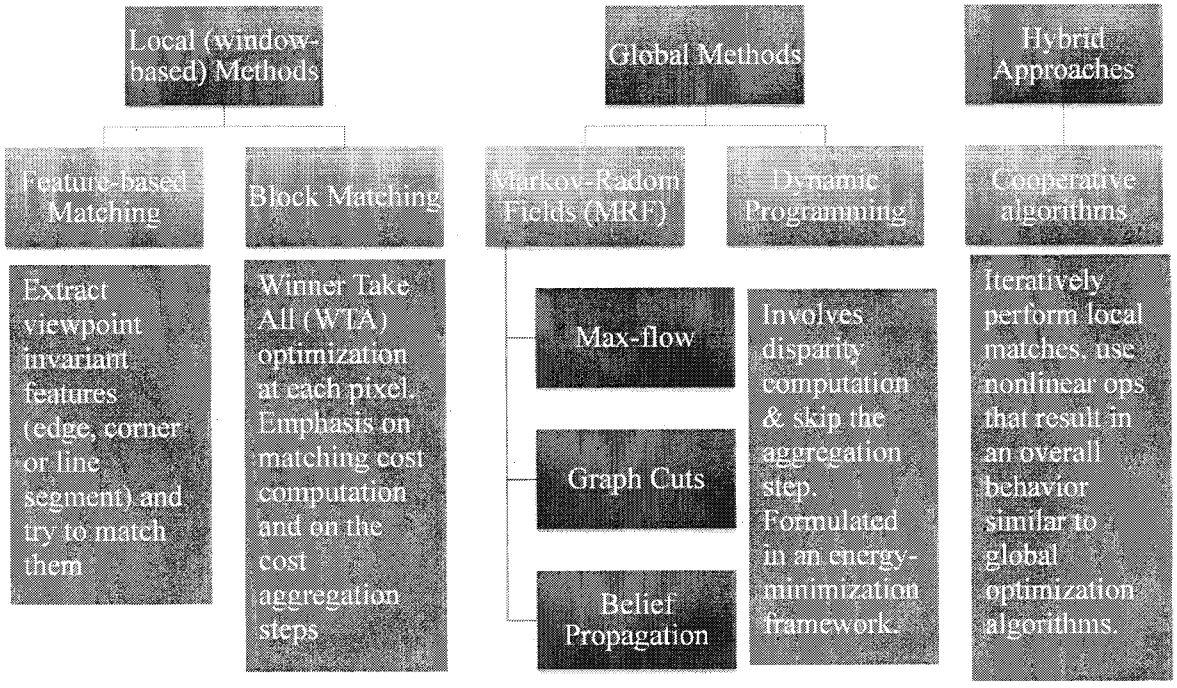


Figure 6. Local, global and hybrid approaches to finding correspondence in stereo-vision.

Stereo-vision has been one of the most heavily investigated areas of research in the field of computer vision. Even though many new algorithms **[3]**, **[4]** introduced in this field focused on the quality and accuracy of the disparity (depth) maps, their successful implementation depends on the complexity of the algorithm and the availability of the hardware platform, which has a direct impact on their suitability for real-time implementation.

Some of the challenges facing the stereo-vision technique include variable illumination, specularities in the scene (non-Lambertian), image noise, camera gain & bias, image sampling and pixelization error, regions that are occluded, low-texture regions, and depth-discontinuous regions.

Some of the examples of real-time, operational stereo vision systems in the literature include systems relying on FPGA **[5]**, commodity graphics **[6]**, PC **[7]**, ASIC **[8]** and hybrids incorporating various processor types **[7]**. The main focus of majority of these systems is on the raw performance, rather than on the size, quality and power consumption.

The aim of the thesis is to develop a fully integrated, small, modular, low-cost, efficient, real-time, miniaturized embedded stereo vision system (MESVS) which should be robust in-terms of radiometric variations, and should be capable of producing high density depth maps to be used for a wide variety of imaging applications in real-world environments.

The thesis is organized as follows. Chapter 2 provides the relevant review of the literature followed by the description of the system design, hardware and software implementation in Chapter 3. The experimental results are presented in Chapter 4, which demonstrates the efficiency and robustness of the MESVS system. Conclusions, along with the suggestions regarding the future areas of development for the system are provided in Chapter 5.

Chapter 2

Literature Review

Stereo vision systems have previously been implemented using huge, complex, custom (expensive) hardware systems [**10**]. Due to the recent advances in the stereo-vision algorithms, an increase of computational power per square inch, exponential increase in speeds, reductions in the size and cost of the processors, it is now possible to implement a stereo-vision system capable of producing accurate, dense depth maps in real-time, which can fit comfortably in the palm of the hand. This chapter provides a review of the existing stereo-vision systems.

Kanade et al [**10**] developed a custom five-camera stereovision machine (CMU machine) with C40 DSP arrays, able to process depth maps at 30 fps with image resolutions of 200× 200. SAD (Sum of Absolute Differences) algorithm was employed with disparity range of 64.

Woodfill et al [**11**] designed a stereo vision system based on 16 Xilinx 4025 FPGAs, and 16 one-megabyte SRAMs. The device generates dense disparity maps at 320× 240 image resolution at video rate, and communicates with PC via PCI bus.

The above systems are quite large in size, expensive to build and operate and are thus not suitable for practical purposes.

SRI's Small Vision Module (SVM) produces dense depth maps at 6 frames per second (fps) on 160x120 images. It consists of two CMOS 320x240 grayscale cameras, low-power A/D converters, a digital signal processor and a small flash memory for program storage. The SVM fits within 2" x 3" (see Figure 7). During operation, the module consumes approximately 600mW power. The software consists of LOG transform of left and right images, followed by area based correlation (absolute differences) with disparity set to 16, post-filtering with an interest operator, left/right consistency check, and range interpolation [**12**].

TYZX DeepSea V2 Stereo Processor embedded in a PCI (Peripheral Component Interconnect) card features a custom ASIC which provides depth data (512x480 image resolution) to a PC (Personal Computer) at 60 fps, consuming <15W power (see Figure 7). It employs census based correlation algorithm. The useful operating range is from 2.7m to 35m, with range resolution of 0.01m @ 3m to 1.1m @ 38m, and spatial resolution of 0.004m @ 3m to 0.05m @ 38m [**13**].

Videre Design's STOC stereo camera (640x480 image resolution) uses local area-based method for computing correlation at roughly 29fps (disparity range of 64, window size of 15x15), consuming about 2.4W power (see Figure 7). It employs Xilinx Spartan 3 – 1000 FPGA running at 84MHz [**14**].

MSVM-III uses three cameras to produce depth maps at 30 fps (640x480 image resolution, 64 disparity levels) [**15**]. An FPGA chip running at 60MHz is employed to compute trinocular rectification, LoG filtering, and area-based matching (see Figure 7).

Bumblebee2 [**16**] is a stereo vision system commercially produced by PointGrey research. It is capable of producing dense depth maps at 48 fps at image resolution of 640x480, consuming 2.5W at12V. The image data is simply streamed over the FireWire port and the processing is done on the PC (see Figure 7).

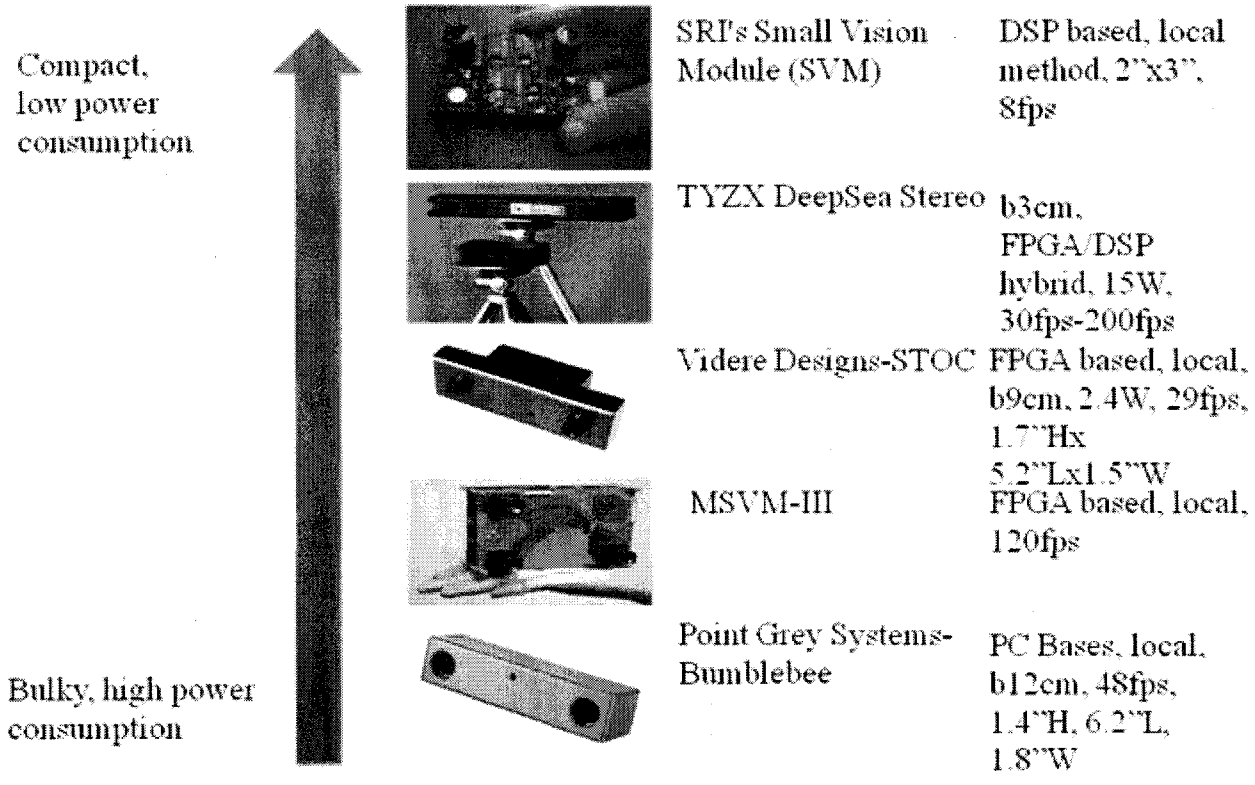


Figure 7. Sample of some existing Stereo-vision systems.[1]

[1] SRI's Small Vision Module (SVM), http://www.ai.sri.com/~konolige/svs/svm.htm; TYZX DeepSea Stereo, http://www.tyzx.com/products/cameras.html; Videre Designs-STOC, http://www.videredesign.com/vision/stoc.htm; MSVM-III [15]; Point Grey Systems-Bumblebee, http://www.ptgrey.com/products/bumblebee2/index.asp

All the above systems are available for commercial purposes and are suitable for practical purposes. Videre Design's STOC camera relies on Laplacian of Gaussian (LoG) image filter for pre-processing input images, followed by sum of absolute differences over a square window to compute correlation. According to the recent studies **[17]**, **[18]**, **[19]** in the presence of local radiometric variations, rank and census transforms have been shown to outperform other methods such as LoG, NCC (Normalized-Cross Correlation), and HMI (Hierarchical Mutual Information) for correlation based matching. TYZX DeepSea stereo and PointGrey's Bumblebee2 systems require a PC to be attached to stereo cameras which does the processing. Thus, they are not suitable for applications that require the system to be compact, power efficient and mobile. MSVM-III requires multi-baseline calibration, rectification and matching which is quite complex in nature. Even though more cameras may yield higher accuracies, they add to the system cost and affect the system size, and power consumption. SRI's SVM module does not produce depth maps at video rate.

Thus, there is a requirement for a fully integrated, small, modular, low-cost, efficient, real-time, embedded stereo vision system that is capable of producing high density depth maps to be used for a wide variety of imaging applications in real-world environments.

The following table summarizes the design specifications for an MESVS system:

Table 1. Design specifications for a Miniaturized Embedded Stereo-Vision System (MESVS)

Specification	**Target Value**
Baseline	<=3cm
Dimensions	LxW <=5cm
Frame rate	>=10fps
Resolution	>QQVGA (160x120)
Power consumption	<3W
Other Features: Produce accurate dense depth maps, Invariant to radiometric variations, Low-cost, Modular design.	

Chapter 3

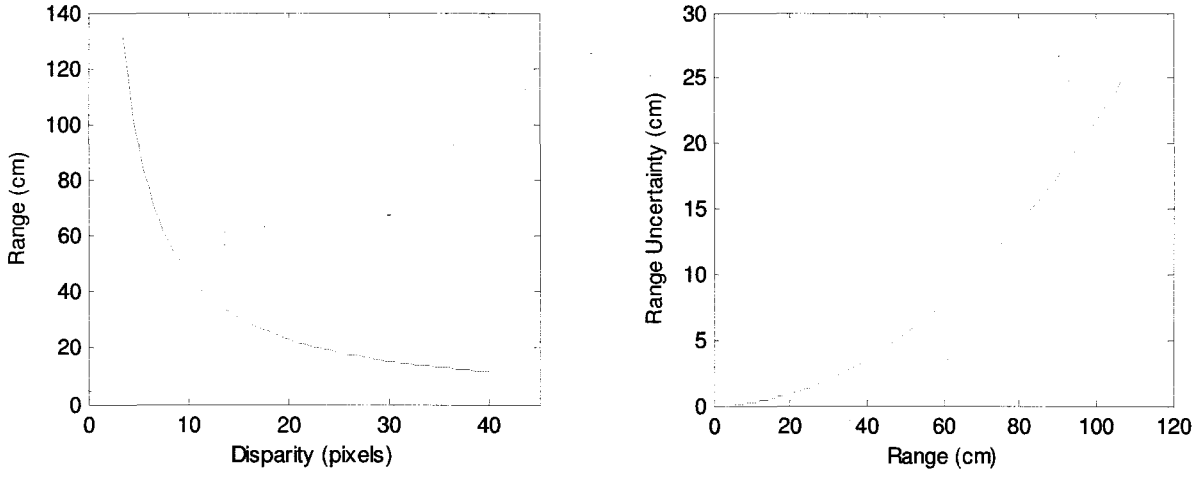
Miniaturized Embedded Stereo-Vision System (MESVS)

3.1 System Overview

Miniaturization of the stereo-vision system is accomplished not only by reducing the overall dimensions of the components used, but also by reducing the baseline, i.e. the distance between the camera's centers of projection. The size of baseline has a direct impact on the range of measurable objects in the scene (Horopter) and the related accuracy (range resolution). As can be seen from (2), as the baseline b decreases, the range r decreases, whereas the range uncertainty Δr increases. Thus, there has to be a trade-off between the horopter and the associated accuracies for disparity values.

$$\Delta r = \left(\frac{r^2}{bf}\right) x \Delta N \qquad \textbf{(2)}$$

where, f is the focal length, x denotes the pixel size and the change in disparity is denoted by ΔN. Figure 8 shows the relationship between range vs. disparity, and range vs. range uncertainty.



(a) (b)

Figure 8. (a) Range vs. Disparity. (b) Range uncertainty vs. Range [20], [19].

To fit the system within 5x5cm, while having an acceptable measurable range and horopter, horopter and disparity ranges have to be carefully chosen. This analysis has to be based on the system's baseline (28mm), focal length (2.8mm), and pixel size (17um). With the horopter set to 5-35, and disparity range of 30 in the above equation, we get an acceptable measurable range of about 15-100 cm. For objects within close vicinity (less than 50 cm) the maximum uncertainty is around 5 cm. Careful analysis shows that using a well adjusted horopter and a large disparity range, the system is capable of compensating for the impact of small baseline and is capable of retrieving range information with acceptable level of accuracy.

3.2 Processor selection

Selecting the desired processor for a computationally intense, stereo-vision application, is of critical importance, as it heavily influences the product cost, performance, and power consumption. There are many types of processors available in the market including PC CPU, Embedded RISC CPU, application processors (like DSPs[1]), media processors (like ASIP[2]), FPGAs (Field Programmable Gate Arrays), and

[1] Digital Signal Processors
[2] Application Specific Integrated Processors

ASSPs (Application Specific Signal Processors). The selection criteria for choosing the proper processor type for the application of stereo-vision include:

a. Performance considerations (like speed, memory handling, data buses, energy consumption, benchmarking results, etc.),
b. Cost of Integration,
c. Availability and roadmap,
d. Development considerations (like single vs. multi-core, number of I/O ports, instruction set architecture, developer familiarity, compatibility, tools-compilers and profilers, support, etc.),
e. Packaging requirements, and,
f. Operating temperature range, among others.

Media processors like ASIPs provide higher performance than most DSPs and GPPs (General Purpose Processors) and have better support for video processing; however, they have complex programming models, higher developmental cost, and higher associated risk as their roadmaps are unclear. FPGAs can be reconfigured dynamically, offer architectural flexibility, high throughput and performance, all resulting in higher efficiency. However, their suitability for low-power, cost sensitive and stereo-vision applications has not yet been proved. ASSPs incorporate one or more processor types that are well matched to the application, and thus offer excellent performance, and energy efficiency; however, ASSPs are often inflexible, have a sharp learning curve and require extensive tuning. Their roadmap is unclear and the benefits of low cost can only be realized when produced in mass-quantities. Application processors offer adequate performance, portability, energy efficiency, integration, and support for video-based applications; however, they are less powerful than other types of processors mentioned above [**21**].

Recently, a new family of Convergent processors (e.g. BlackFin by Analog Devices) has emerged that is ideal for advanced video processing applications, combining both MCU (Micro Controller Unit) and DSP functionality into a single device with a unified architecture, high clock speed, low power dissipation per unit of processing, smaller form factor and flexible programming model [**22**]. They function simultaneously as a 16-bit DSP and a 32-bit MCU while supporting both DMA (Direct Memory Access) and cache functionality. Embedded system programmers leverage the portability of the code written in C and try optimization approaches at the algorithm level and compiler level. However, in-order to achieve real-time performance, assembly language coding is required.

Convergent processors allow the developers to create applications in C/C++, as the processor is optimized not only for computation on real-time multimedia data, but also for control tasks. The benefits

include: best utilization of existing skill sets within a team, reduced time to market and lifecycle costs, higher processing speeds, and ease of maintenance. Thus, we have chosen the dual core, BlackFin processor (ADSP-BF561) as our processing platform for the stereo-vision system. More details on the hardware and memory architecture can be found in the following sections.

3.3 Hardware Implementation

The system hardware consists of a state-of-the-art embedded processor ADSP-BF561, two tiny CMOS camera sensors, two Parallel Peripheral Interface (PPI), 64MB of SDRAM, 8MB of addressable flash memory, JTAG interface, and SPI port. Figure 9 shows a high-level block diagram of the system.

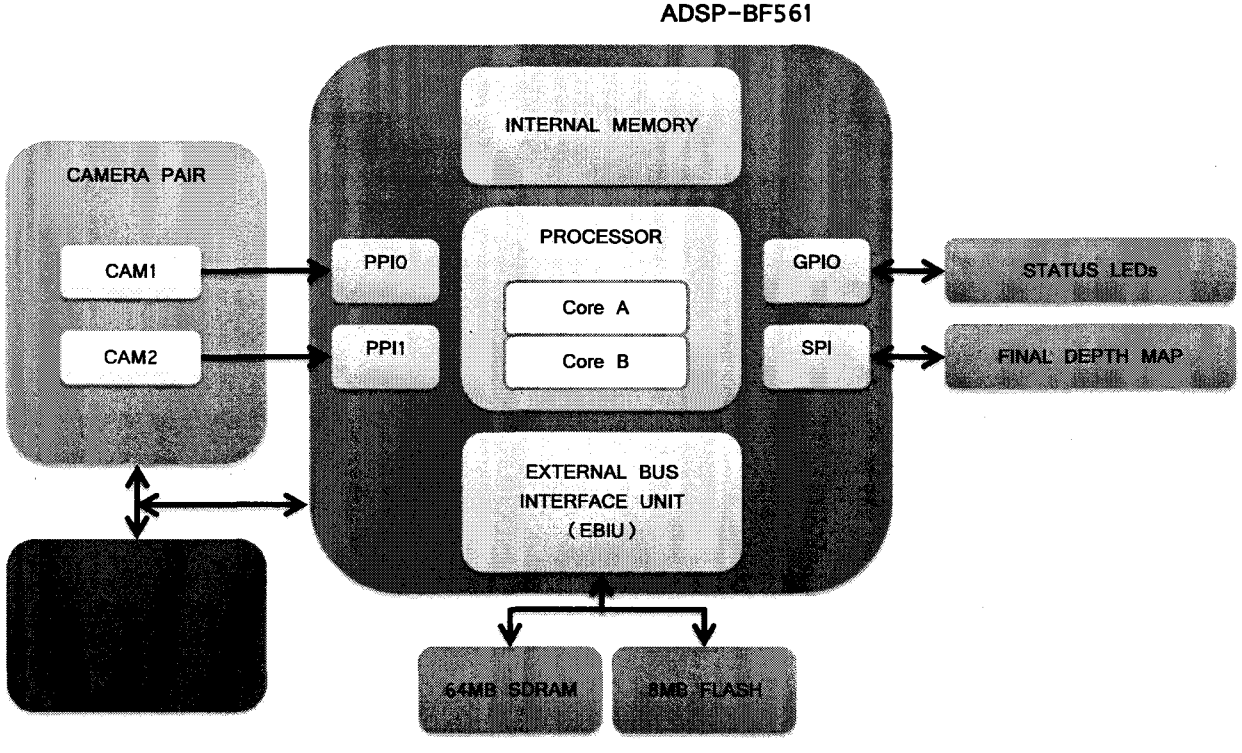


Figure 9. System block diagram

ADSP-BF561 is one of the latest members of BlackFin family of embedded processors featuring a dual core processor, with each core capable of 1200 MMACs@600MHz (2400 MMACs total). ADSP-BF561 provides the best compromise between performance and size versus cost and power consumption [**20**]. It also possesses two flexible video ports used to capture stereo image pairs from CMOS cameras. Table 2 provides a summary of ADSP-BF561 specifications.

Table 2. Summary of specifications for BlackFin processor.

Clock Speed (MHz)	600MHz (per core)
MMACS (Max)	2400
RAM Memory (kBytes)	320
External Memory Bus	32bit
PPI ports	2
Core Voltage (V)	0.8-1.2
Packages	297-PBGA, CSP_BGA

3.3.1 Memory Architecture

Performance of an embedded system is directly dependent on how the memory and data is managed. BlackFin processors support a modified Harvard architecture along with a hierarchical memory structure. Figure 10 shows an overview of the memory architecture of our system. Each BlackFin processing core has access to the high-speed, high-performance, low-latency, Level 1 (L1) memory that typically operates at the processor speed. L1 memory is made up of 64KB of data memory and 32KB of instruction memory. Memory Management Unit (MMU) defines the properties of a given memory space and protects the system registers from unintended access. A unified Level 2 (L2) memory shared by both of the cores operates at approximately half of the core-clock speed, resulting in a higher latency compared to L1 memory. External memory also called Level 3 (L3) memory consists of 64MB of SDRAM and 8MB of flash memory. The system's firmware is burnt onto the flash memory. Although L3 memory is quite large, the access time is measured in System Clock Cycles (SCLK), which is usually much less than the CCLK rate.

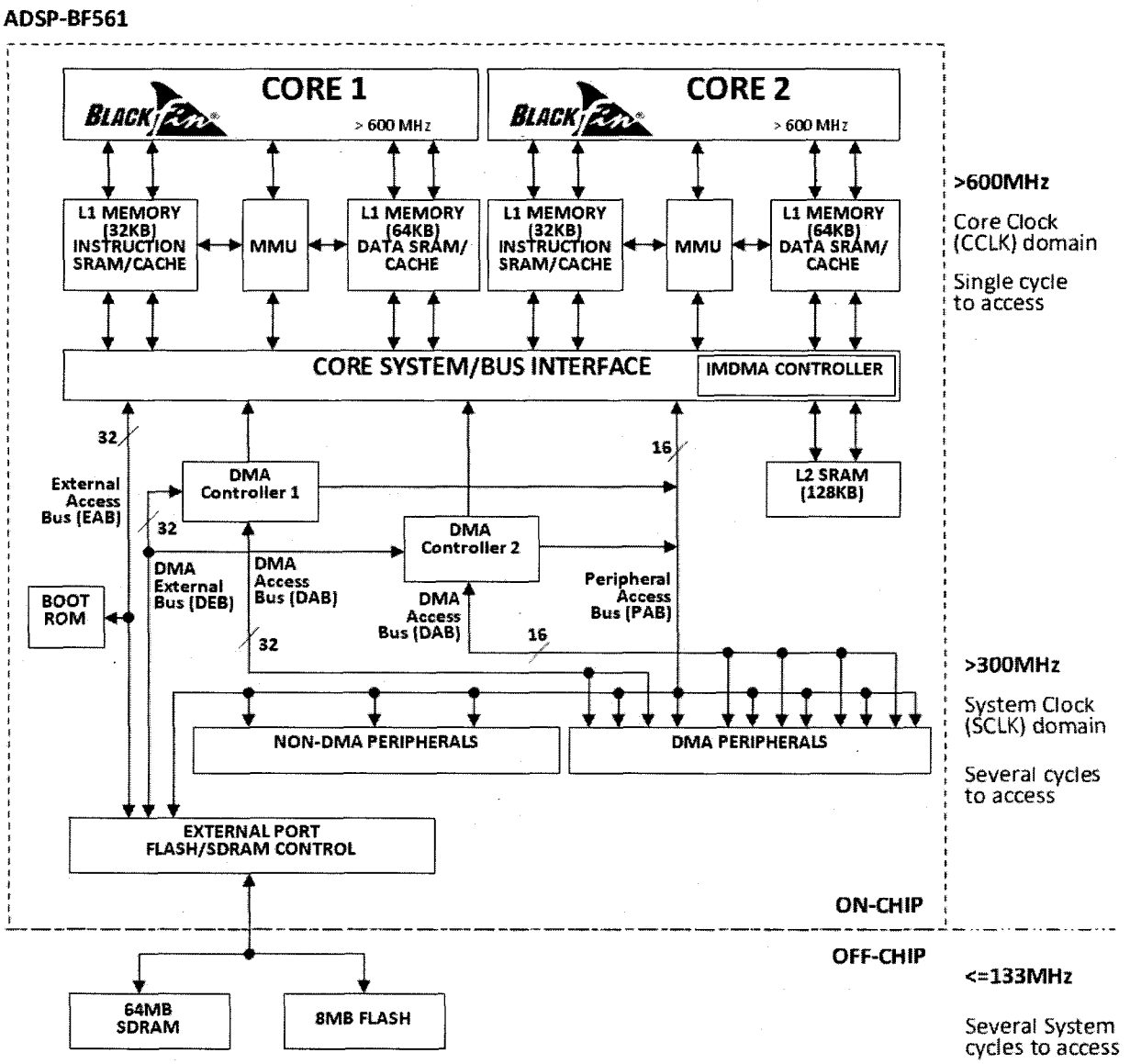


Figure 10. Memory architecture of MESVS system.

Multiple DMA channels facilitate data movement between the peripherals and the memory systems, with no overhead on the processor. Extensive deployment of DMA functionality has allowed us to improve system performance and reduce the run-time of the matching algorithm, and rectification, as explained in section 3.5.

3.3.2 CMOS vs. CCD Sensors

Digital cameras use solid state image sensor which contain millions of photosensitive diodes called photosites. These sites integrate the intensity of the light falling on the diodes by accumulating the charge during the brief moment the shutter is open. A charge-coupled device or CCD utilizes an analog-to-digital converter to convert the charge to a digital number. The charges are coupled in such a way that only one row can be read at a time.

In most CMOS (Complementary Metal-Oxide Semiconductors) devices, transistors are utilized to enable reading of individual pixel's intensity. Each pixel has its own charge-to-voltage conversion, and the sensor often also includes amplifiers, noise-correction, and digitization circuits, so that the chip outputs digital bits. This increases the complexity, reduces effective area to capture light and makes the system non-uniform. However, additional on-chip features can be added at little or no extra cost, which may include image stabilization and image compression. This in essence, makes the camera lighter, smaller and thus cheaper to produce. It also requires very low power consumption. Integrating these features in a CCD camera will make the manufacturing process so complex, that it would be un-economical to produce. The best performance on the CMOS cameras can be obtained in an outdoor environment, as they suffer in low-light conditions.

The percentage of a pixel devoted to collecting light is called the pixel's fill factor which has a direct correlation with the sensitivity of the sensor and is inversely correlated with the exposure time. CCDs have a 100% fill factor but CMOS cameras have much less. To increase the fill factor for CMOS image sensors, micro-lenses are typically integrated in the package.

A detailed comparison between the CCD and CMOS image sensors is provided in Table 3.

Table 3. Comparison of CCD vs. CMOS image sensors.

	CCD (Charge Coupled Device) Sensors	**CMOS (Complementary Metal-Oxide Semiconductors) Sensors**
Cost	Expensive to produce-special manufacturing methods employed	Inexpensive to produce-semiconductor technology
Power	Consumes up to 100x more power than CMOS	Low power consumption
Noise	High quality, low noise images	Susceptible to noise; lower performance in low light conditions
Maturity	Produced for longer period, higher quality images, more pixels	Less mature
Extended Functionality	Technically feasible, other chips are used	Other circuitry easily incorporated on same chip
Fill Factor (percentage of a photo-site that is sensitive to light)	high	low

Taking into account all the above factors, we have chosen to go along with the OmniVision's 7660 CMOS image sensors.

Table 4 below, shows the main specifications of the chosen image sensor.

Table 4. Specifications of OmniVision's OV7660 Camera module

Feature	**Value**
Output Formats (8-bit)	YUV/YCbCr 4:2:2 ; RGB 4:2:2; Raw RGB Data
Lens Size	1/5"
Frame Rate	1.0V/Lux-sec

S/N Ratio	>48dB (AGC off, Gamma=1)
Dynamic Range	>72dB
Scan Mode	Progressive
Electronic Exposure	Up to 510:1 (for selected fps)
Pixel Size	4.2um x 4.2um
Fixed Pattern Noise	< 0.03% of $V_{PEAK-TO-PEAK}$
Package Dimensions	6.5mmx6.5mmx4.84mm

3.3.3 Design Considerations

Electrical wiring or traces on the printed circuit board (PCB) can be thought of as multiple charge carrying conductors q_1,q_2, separated by a distance r. According to Coulomb's law, the force of attraction F can be given as:

$$F = k(\frac{q_1 q_2}{r^2}) \tag{3}$$

where, k is the coulomb's constant which depends on the properties of the space. It can be given as:

$$k = \frac{1}{4\pi\varepsilon} \tag{4}$$

where, ε is the permittivity of dielectric, which can be calculated as:

$$\varepsilon_r = \frac{\varepsilon}{\varepsilon_0} \tag{5}$$

where,$\varepsilon_r$is the dielectric constant, and $\varepsilon_0$is the permittivity of empty space.

The force of attraction between the conductors adds to the noise, ground bounce and crosstalk. The construction materials used for PCBs have a set dielectric constant, the most common amongst those is Fr-4/glass with a dielectric constant of 4.1 and a loss tangent value of 0.019@1MHz [22]. The larger the loss tangent, the higher is the absorption of high frequencies by the dielectric material; the higher is the attenuation of signals. Di-electric constant has an effect of the impedance of transmission line (signal traces), where lower value permit faster propagation velocities v_p as shown by the equation below:

$$v_p = \frac{C}{\sqrt{\varepsilon_r}} \tag{6}$$

where, C is the speed of light ($3x10^8$m/s). The propagation delay t_{pd} for a length l of the conductor can be given as:

$$t_{pd} = \frac{l}{v_p} \tag{7}$$

From the above equation, it can be clearly seen that as the length of the conductor increases, propagation delay increases.

Two long length parallel traces also have a mutual capacitance between them. Changes in the voltage in one conductor causes the capacitance to change, resulting in cross-talk. Magnetic field from one trace can induce a signal in another trace, commonly referred to as mutual inductance. Due to mutually inductive effects, care should be taken in routing clock signals. The traces should be as straight as possible. Placing a ground plane next to the clock output minimizes noise. To avoid ringing caused by reflection on the transmission line, the impedance of the source (Z_S) must equal the impedance of the trace (Z_0), as well as the load (Z_L).

At low frequencies, current flows through the path of least resistance. At high frequencies, the current flows through the path of lease inductance. Due to proximity effect, at higher frequencies, the current returns to the ground plane. The current does not flow through the entire cross-section of the conductor, but flows on the surface of the traces. As the depth of the trace changes, current density changes. This phenomenon is referred to as skin depth. It is the depth at which the losses associated with a conductor carrying AC equals those of the hollow conductor carrying DC signals. The thicker the conductor, more pronounced is the skin effect.

As the frequencies increase, the rise time decreases causing high transient currents in outputs as they discharge load capacitances. Switching of many signals from high-to-low-to-high may cause a board-level phenomenon known as ground bounce.

Every PCB generates electromagnetic interference (EMI) which is directly proportional to the change in current or voltage with respect to time. Minimizing crosstalk and proper grounding can significantly reduce EMI issues. Spacing between the signal lines should be widened as much as possible. Single ended signals should be routed on different layers of the PCB and should be orthogonal to each other. Decoupling capacitors should be added for VCC/GND pairs, and should be placed closed to the power pins. Wherever possible external pull-up resistors should be eliminated and replaced by pull-down resistors. Low effective series resistance (ESR) capacitors of ESR < 400 mΩ should be used as decoupling capacitors.

3.3.4 PCB design flow

The design of a successful PCB layout starts with the schematic design followed by creating a Bill of Materials (BOM), design review, ordering of parts, generating design rules, component placement, routing, generation of silkscreen, assembly & fabrication drawings, followed by the actual fabrication, assembly and testing. Figure 11 shows the typical design flow:

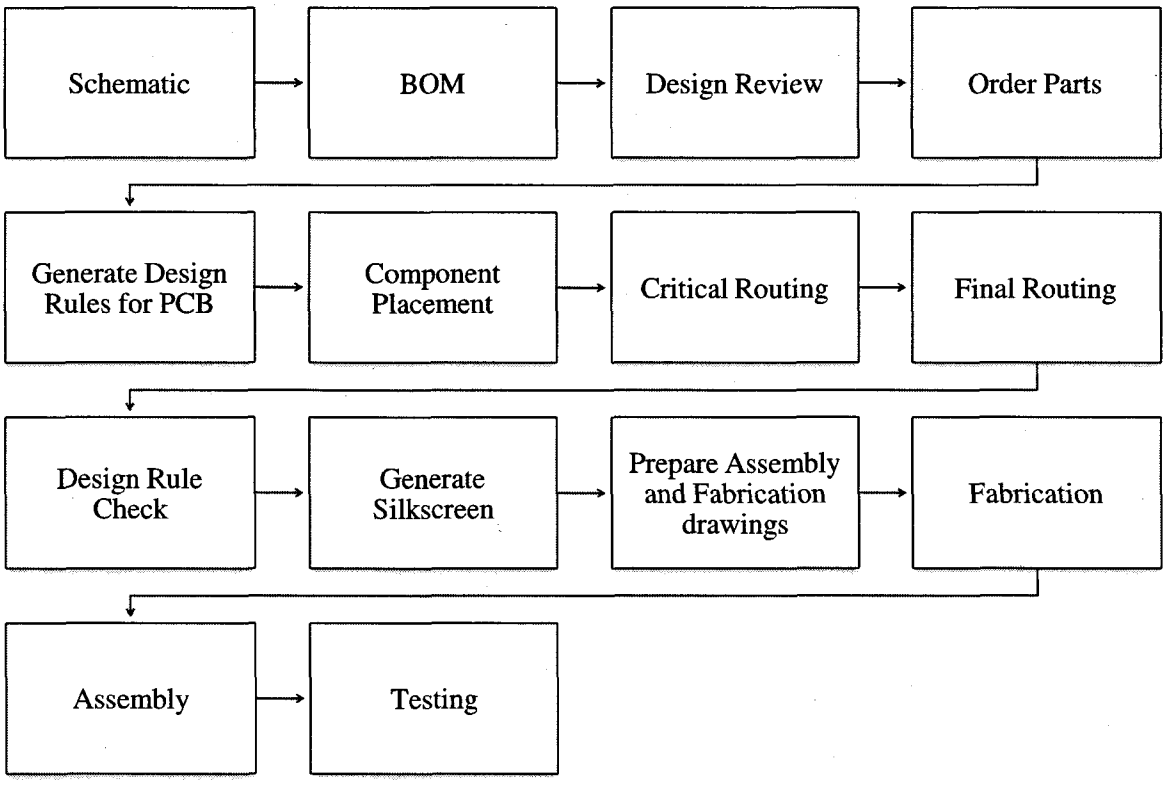


Figure 11. Hardware design flow.

3.3.4.1 Schematic capture

This involves capturing the electronic design i.e. how components are connected to each other (net-list). Package types of all the parts are created, usually a long, laborious process.

3.3.4.2 Bill of Materials (BOM), Design Review and Ordering Parts

Once the interconnection between the components is defined by the net-list, a bill of materials can be easily constructed defining the symbols used, type and description of parts, number of pins in the components, package types, dimensions and effective areas, distributor & manufacturing part numbers, unit prices, total quantities and effective total price. Table shows the breakdown of the costs of components used in MESVS module.

Table 5. Breakdown of costs of components used in MESVS module.

PART TYPE	PRICE
Processor	$295.96
Cameras	$121.42
Resistors	$8.19
Capacitors	$0.50
Voltage Regulators	$2.32
Oscillators	$5.66
Switches	$0.93
Connectors	$23.55
Diodes	$1.23
TOTAL COST	*~$459.76 +Tax + Shipping*

As can be seen from the table above, the most expensive components include the processor, cameras, and connectors. A review of the design is carefully conducted followed by ordering of the parts.

3.3.4.3 Design Rule Check (DRC)

All design parameters that are relevant to the PCB design and manufacturing are specified as design rules. Some of the parameters include number of routing layers, copper thickness, isolation thickness, minimum

size, clearances, distances & drill sizes between objects in the signal layers, size of pads, vias (plated through holes-PTH used to provide electrical connection between a trace on one layer of the to a trace on another layer) & micro-vias, restring width, and mask. Most PCB design software includes a bot (automated piece of software) which ensures that the design conforms to generally acceptable rules. They are often limited in scope, as they usually cannot detect errors such as mirrored layers, drill file problems, missing layers, or if traces are connected to the wrong pins on an IC. Thus, the check has to be then conducted by a human to detect defects, and catch errors at an early stage, thereby making sure high quality fabrication result.

3.3.4.4 Generate Silkscreen and prepare assembly and fabrication drawings

The PCB layout has to be submitted in the RS274X gerber format, which captures detailed information about the design. Most software output the following files as an output:

- Top Copper Layer (*.cmp or *.gtl)
- Top Solder Mask (stc or gts)
- Top Silkscreen (plc or gto)
- Bottom Copper Layer (sol or gbl)
- Bottom Solder Mask (sts or gbs)
- Bottom Silkscreen (pls or gbo)
- Drill File (drd or txt)

Silk-screens are colored marks (usually white) on the PCB board to identify components for later assembly and troubleshooting processes.

3.3.4.5 PCB Fabrication

Fabrication of a PCB typically involves many steps, starting with the drilling of the copper laminate followed by the deposition of Cu onto the drill board. The next step involves optical lithography, which transfers the desired pattern (PCB layout) onto the photo-resist substrate using a photo mask that is sensitive to light. This step is repeated several times, until the PCB layout is successfully transferred onto the surface. Then the substrate is plated or mechanically polished, followed by etching process. Etching process utilizes acids, bases or other chemicals to dissolve unwanted materials from the substrate thereby leaving the desired routing layout. Once the traces are clearly visible, Hot-Air-Solder-Leveling is used to apply solder to the circuit board based on the solder mask, and flux is applied to the boards.

Electrical conductivity tests can be performed at this stage using automated machines, followed by V-scoring to panelize the boards, if needed. PCB boards are then cleaned, packaged and shipped.

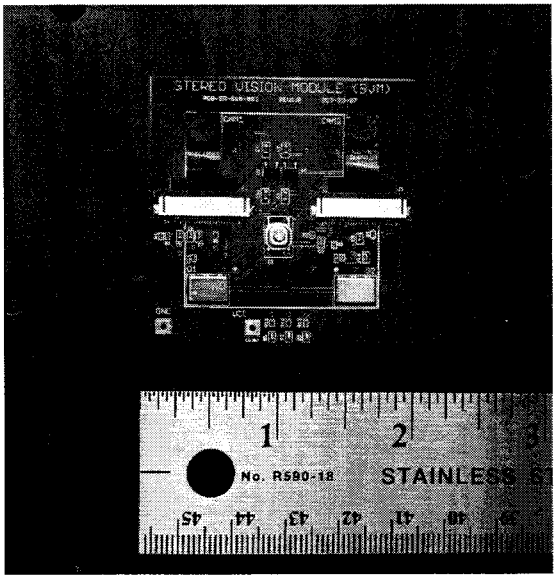
3.3.4.6 PCB Assembly

Since the boards use surface mount (SMT) technology, one of the hardest and most expensive of all steps is the actual assembly of the components on top of the PCB. Components are placed by automated machines onto the board, which is followed by visual inspection to detect missing or misaligned components and solder bridging. If needed, rework is done by manual intervention.

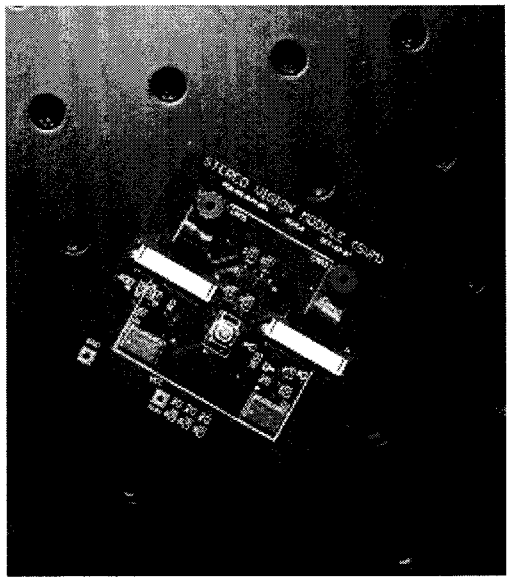
3.3.4.7 PCB Testing

From design to fabrication to assembly, errors may get accumulated over a period of time. A PCB has to be inspected and then tested to ensure that the boards work accordingly. Some of the tests include AOI (Automated Optical Inspection), X-Ray Testing, In-Circuit Testing, Flying Probe Testing and JTAG Testing. For more details, please refer to [**24**].

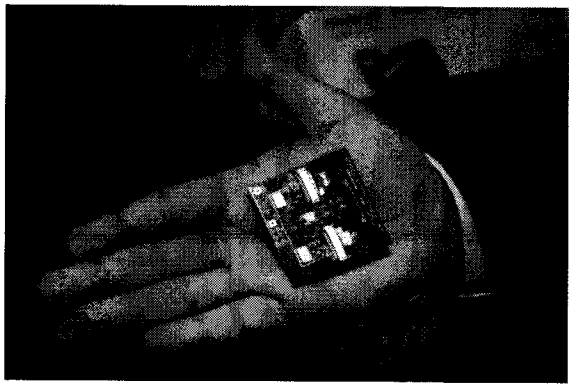
Figure 12 shows the final Miniaturized Embedded Stereo Vision System (MESVS).



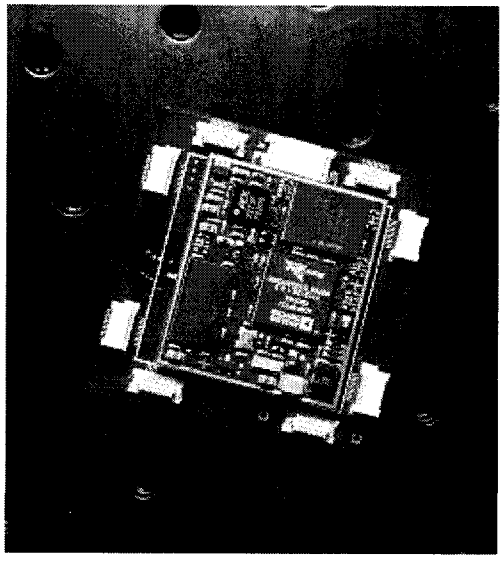
Shown with a ruler.



Front side



Shown in hand



Back side

Figure 12. Miniaturized Embedded Stereo Vision System (MESVS).

3.4 Software Algorithms

MESVS relies on an efficient and robust local stereo matching engine to retrieve the depth information of the scene in real-time (i.e. about 20 fps). As illustrated in Figure 13, there are five major stages that constitute the stereo matching engine, namely, image acquisition and sub-sampling, stereo rectification, pre-processing, matching and left/right consistency (LRC) check, and post-processing.

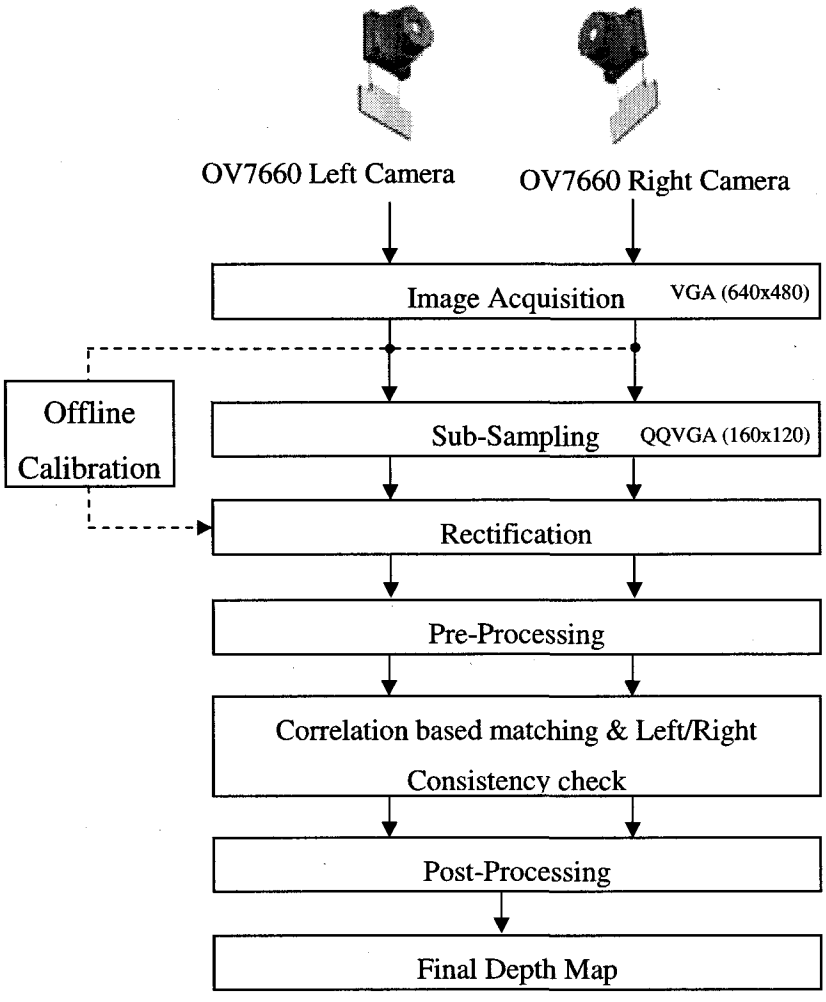


Figure 13. Overview of the stereo-matching engine

3.4.1 Image Acquisition, Offline Camera Calibration and Sub-Sampling

CMOS (Complementary Metal-Oxide Semiconductor) camera technology has been chosen for the module, as the camera modules cost less than the CCD (Charge Coupled Device) based imagers, consume relatively lower power, and can incorporate other circuitry on the same chip (like clock drivers, timing logic, signal processing, etc.). Images are captured by OmniVision's OV7660 camera modules, in VGA resolution, and thus require about 300KB of storage space, per image pair. An offline stereo camera calibration is performed initially to obtain both intrinsic and extrinsic parameters of the system, using the

MATLAB toolbox from the Institute of Robotics and Mechatronics at Caltech [**25**]. This step helps in rectification of the images and if needed, reconstruction of world views.

In the calibration process, the main objective is to find the intrinsic quantities that affect the imaging process like position of the image center, focal length, scaling factors for row and column pixels, skew factor, and lens distortion among others; and extrinsic parameters such as rotation and translation of the camera. Images of a planar checker board target as captured by the camera are loaded into the toolbox (see Figure 14), which is then followed by running corner extraction, and the main calibration engine. If needed, accuracies are controlled, and images added, suppressed or un-distorted, to obtain the final calibration parameters.

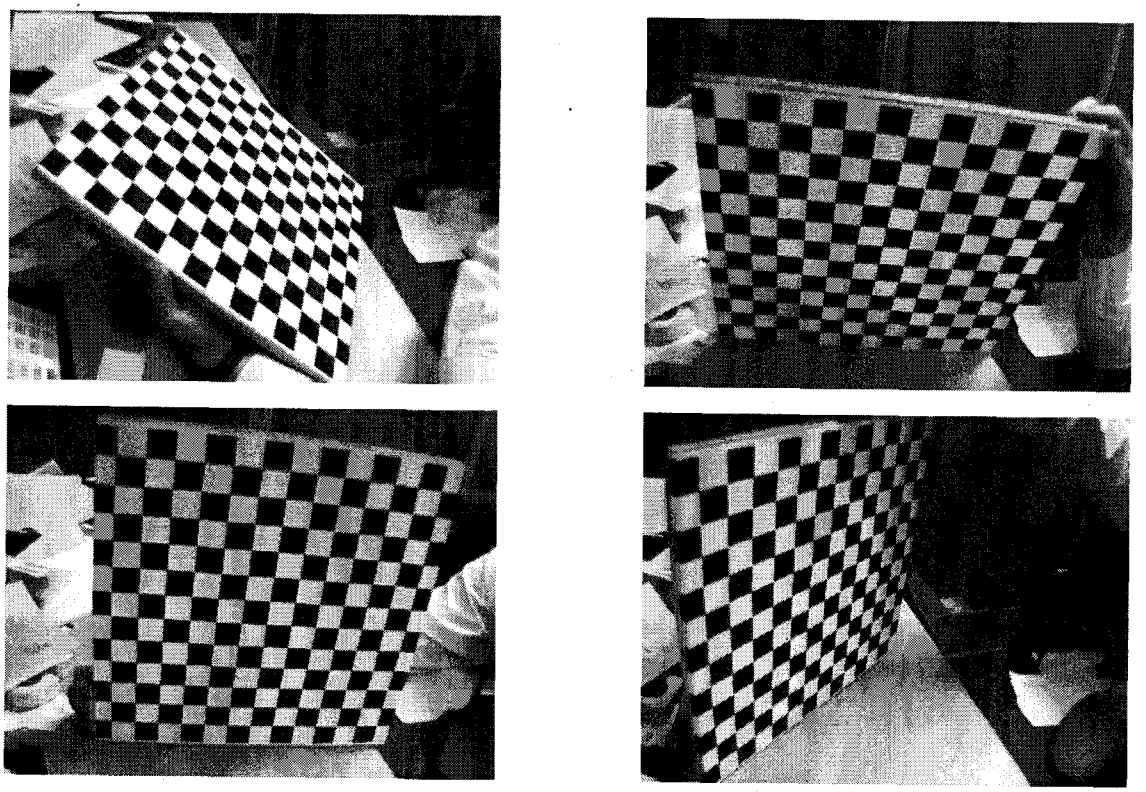


Figure 14. Sample of images of the planar checkerboard target used for calibration [25].

Following figure shows a sample spatial configuration of the two cameras and the calibration planes as viewed from the toolbox:

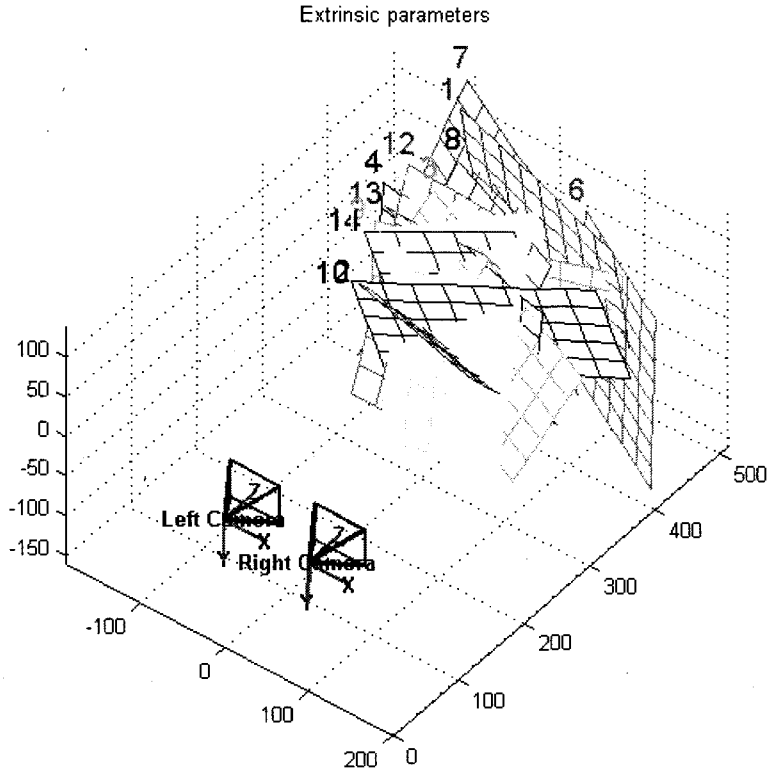


Figure 15. Sample spatial configuration of the two cameras and the calibration planes [25].

Based on our experimentation, tangential distortion removal step was not implemented, as it leads to only a slight improvement in the disparity result, yet considerably increases the computational complexity of the un-distortion process.

The images that are captured, cannot be stored directly in the internal memory of the processor (due to the limited amount of fast, on-chip L1 memory available), but can be stored in external memory; however, the external memory typically operates at a much lower speed compared to the operating speed of the core processor(s). Therefore, there are significant delays associated with external memory accesses, which would reduce the overall system performance. To overcome this issue, and maximize the efficiency, the images are sub-sampled into QQVGA (160x120) resolution, in order to fit within the on-chip memory of the BlackFin® processor. This is accomplished, by skipping every three rows and columns of the input image utilizing the 2D-DMA facility provided by the BlackFin® Processor's Internal Memory DMA (IMDMA) controller (see Figure 16).

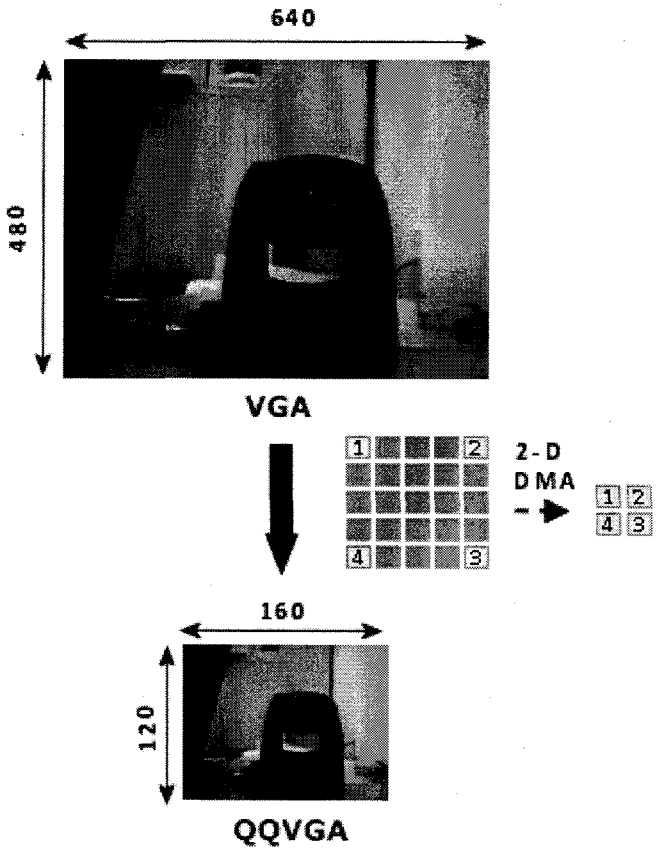


Figure 16. Sub-sampling VGA images to QQVGA images by skipping 3 rows and columns.

3.4.2 Rectification

The images are rectified to make the pairs of conjugate epipolar lines collinear and parallel to the horizontal image axis. This reduces the 2D correspondence problem to a simpler 1D search.

Rectification takes the following intrinsic and extrinsic calibration parameters generated by the previous stage as an input:

```
Input Left Image (var: leftImage),
Input Right Image (var: rightImage),
Focal Length of Left Camera (var: fc_left),
Principal point of Left Camera (var: cc_left),
Skewness in the left camera (var: alpha_c_left),
Distortion in the left camera (var: kc_left),
```

```
Focal Length of Right Camera (var: fc_right),
Principal point of Right Camera (cc_right),
Skewness in the Right camera (var: alpha_c_right),
Distortion in the right camera (var: kc_right),
Rotation Vector (var: om),
Translation Vector (var: T),
Number of rows (var: nx),
Number of columns (var: ny).
```

and, generates as an output the left and right rectified images.

Rectification step includes: back projection, distortion removal, and bi-linear interpolation of pixels, which requires floating-point operations; however the selected embedded media processor can only perform fixed point operations.

Floating point operations can be emulated in software using the available library function calls (implemented in C) which can be inefficient. To ensure the optimum performance of the rectification algorithm, we have implemented a relaxed and fast floating-point routine in assembly language. Wherever possible, faster fractional data type operations (natively supported by BlackFin®) were used instead of floating-point operations, to further improve the efficiency of the algorithm.

Figure 17 displays the left and right rectified images obtained from our module (MESVS-II).

Left Rectified Image

(I_{L_RECT})

Right Rectified Image

(I_{R_RECT})

Figure 17. Left and Right rectified images belonging to the Mug dataset obtained from MESVS-II.

Details on the implementation of the rectification stage can be found in Appendix A.1-A.8.

3.4.3 Pre-Processing

Radiometric variations pose a significant problem in creating a high quality depth map. These variations can be caused by the variable settings (both internal and external) of the two cameras (such as gain, vignetting effect, etc.), image noise, properties of the light source (such as strength, position, orientation, etc.), and properties of the objects in the environment (such as Lambertian or non-Lambertian surfaces, etc). In-order to reduce the sensitivity of the matching algorithm, and increase its robustness to radiometric variations, we employ a pre-processing stage.

Non-parametric local transforms such as rank and census compute the relative ordering of the intensity values within a neighborhood to reduce the effects of variations caused by camera's gain and bias, and increase the robustness to outliers that typically occur near depth-discontinuities [**26**]. Both of the two transforms are suitable for fast hardware implementations [**27**].

3.4.3.1 Rank Transform

Rank Transform *R(P)* is a form of non-parametric local transform (i.e. relies on the relative ordering of local intensity values, and not on the intensity values themselves) used in image processing to rank the intensity values of the pixels within a square window *N(P)*. The centre pixel's intensity value *I(P)* is replaced by its rank amongst the neighboring pixels *I(P')*[**1**] as shown in the following equation:

$$R(P) = \|\{P' \in N(P) | \, I(P') < I(P)\}\| \tag{8}$$

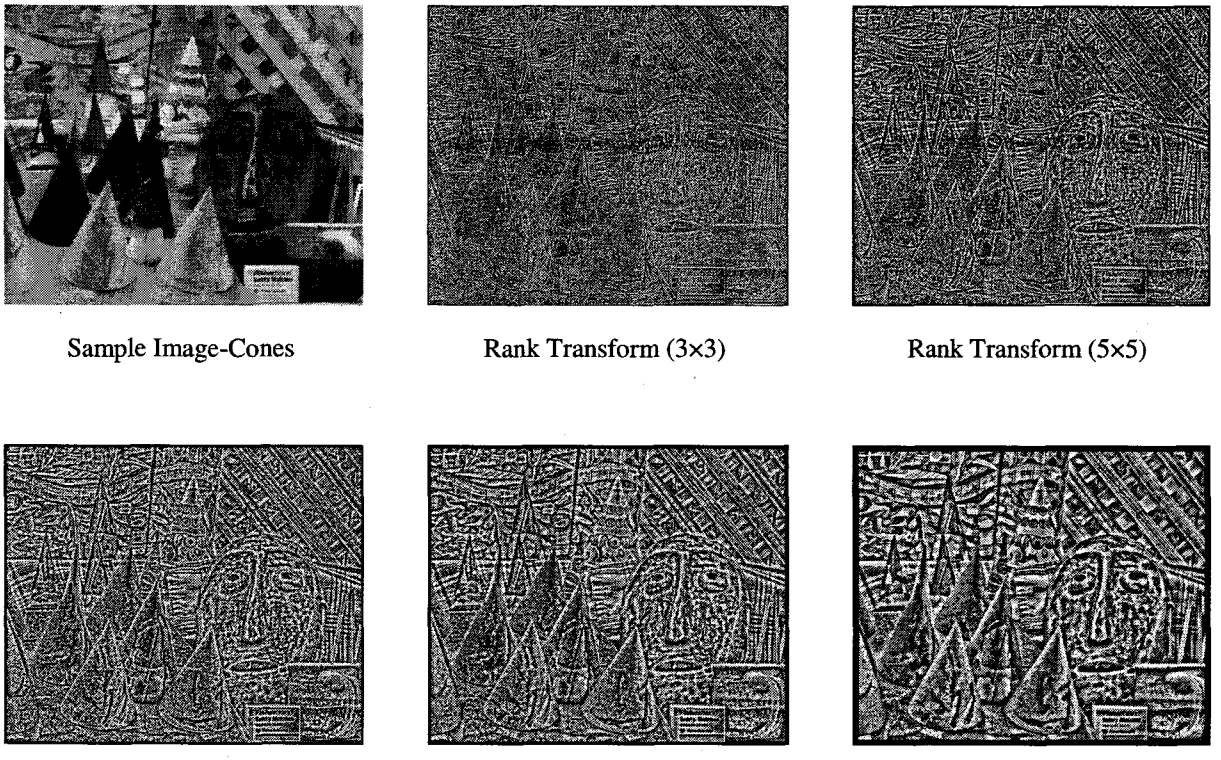
Advantages:

- Reduces effects of variations caused by camera's gain and bias.
- Increase in robustness to outliers near depth-discontinuities.

Disadvantages:

- Loss of information associated with the pixel.
- Rank transform cannot distinguish between rotations and reflections, and has been shown to produce the same rank for variety of patterns [**28**].

The following figure shows the Rank Transform of various window sizes being applied to a sample of image-Cones from 2003 dataset:



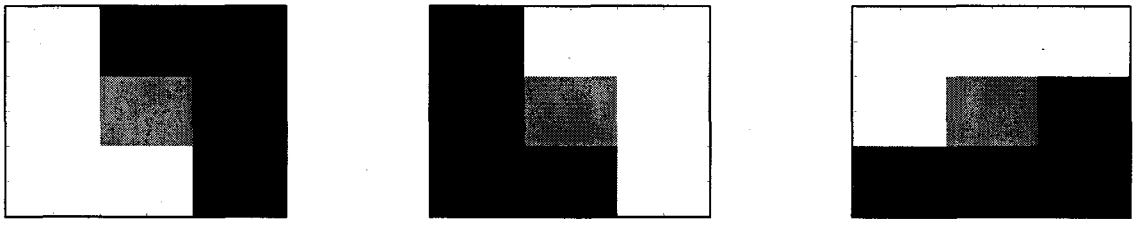
Sample Image-Cones | Rank Transform (3×3) | Rank Transform (5×5)

Rank Transform (7×7) | Rank Transform (9×9) | Rank Transform (15×15)

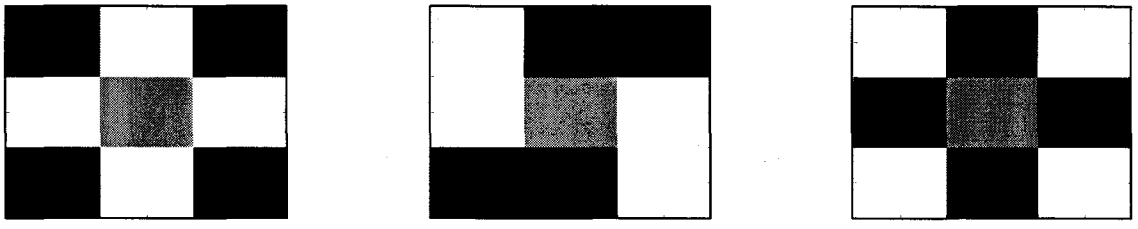
Figure 18. Rank Transforms of a sample image-Cones. Shown from left-right are Rank transforms with square window sizes 3x3, 5x5, 7x7, 9x9 and 15x5.

As can be seen from the above figure, Rank Transform loses color data and information about the local image structure. As the window size increases, the edges become clearer, however, the time required to process the image also increases.

In-order to explore the sensitivity of the Rank Transform to rotations and reflections, test patterns were generated, as shown in the figure below [**28**]:



Test Pattern 1 Test Pattern 2 Test Pattern 3



Test Pattern 4 Test Pattern 5 Test Pattern 6

Figure 19. Test patterns used to explore the sensitivity of Rank Transform to Rotations and Reflections

After applying Rank Transform to the above test patterns, it was found that all of the patterns yielded a rank result of 4, thus proving that Rank transform cannot distinguish between rotations and reflections. More of these patterns can be generated to emphasize the point.

The code used to produce the above rank transforms can be found in Appendix A.16.

3.4.3.2 Census Transform

Census Transform $R_T(P)$ is another form of non-parametric local transform (i.e. relies on the relative ordering of local intensity values, and not on the intensity values themselves) used in image processing to map the intensity values of the pixels within a square window to a bit string, thereby capturing the image structure [**26**]. The centre pixel's intensity value is replaced by the bit string composed of set of Boolean comparisons such that in a square window, moving left to right,

```
If (CurrentPixelIntensity<CentrePixelIntensity) boolean bit=0
else boolean bit=1
```

The mathematical representation of this transform is given by the following equation:

$$R_T(P) = \bigoplus_{[i,j]\in D} \xi(P, P + [i,j]) \tag{9}$$

where, $\oplus$ is the concatenation operator on a set of pixels with a set of displacements D within a square window, and $\xi(P, P')$ is equal to 1 when $I(P') < I(P)$ and 0 otherwise.

For each set of comparisons the bit is shifted to the left, forming an 8 bit string for a census window of size 3×3 and a 32 bit string for a census window of size 5×5.

Advantages:

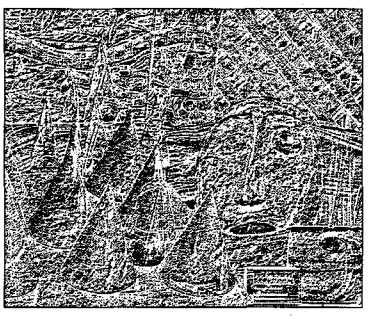
- Reduces effects of variations caused by camera's gain and bias.
- Increase in robustness to outliers near depth-discontinuities.
- Encodes local spatial structure.
- Tolerant to factionalism (If a minority of pixels in a local neighborhood has a very different intensity distribution than the majority, only comparisons involving a member of the minority are affected).
- It can distinguish between rotations and reflections

Disadvantages:

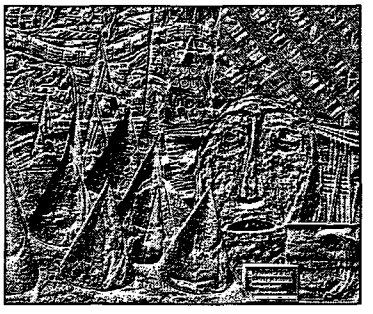
- Loss of information associated with the pixel.

The following figure shows the Census Transform of window size 3x3and 5x5 being applied to a sample of image-Cones from 2003 dataset:

Sample Image-Cones



Census Transform (3x3)



Census Transform (5x5)

Figure 20. Census Transforms of a sample image-Cones. Shown from left-right are Census Transforms with square window sizes 3x3 and 5x5.

The code used to produce the above rank transforms can be found in Appendix A.17.

3.4.4 Correlation based matching and Left-Right Consistency (LRC) check

One of the most computationally intense parts of the stereo vision system is correlation based matching and left-right consistency check. Correlation based matching typically produces dense depth maps by calculating the disparity at each pixel within a neighborhood. This is achieved by taking a square window of certain size around the pixel of interest in the reference image and finding the homologous pixel within the window in the target image, while moving along the corresponding scanline. The goal is to find the corresponding (correlated) pixel within a certain disparity range d ($d \in [0,....,d_{max}]$) that minimizes the associated error and maximizes the similarity. In brief, the matching process involves computation of the similarity measure for each disparity value, followed by an aggregation and optimization step. Since these steps consume a lot of processing power, there are significant speed-performance advantages to be had in optimizing the matching algorithm.

The images can be matched by taking either left image as the reference (left-to-right matching, also known as direct matching) or right image as the reference (right-to-left matching, also known as reverse matching). Classical similarity measures are listed in the following table:

Table 6. Classical similarity measures SAD, SSD, and NCC, along with their corresponding equations. *(i,j)* is the coordinate of the pixel in the square neighborhood window *W*. I_1 is the reference image, I_2 is the target image.

Sum of Absolute Differences (SAD)	$\sum_{(i,j)\in W} \lvert I_1(i,j) - I_2(x+i, y+j)\rvert$
Sum of Squared Differences (SSD)	$\sum_{(i,j)\in W} (I_1(i,j) - I_2(x+i, y+j))^2$
Normalized Cross Correlation (NCC)	$\frac{\sum_{(i,j)\in W} I_1(i,j).I_2(x+i,y+j)}{\sqrt[2]{\sum_{(i,j)\in W} I_1^2(i,j).\sum_{(i,j)\in W} I_2^2(x+i,y+j)}}$
Sum of Hamming Distances (SHD)	$\sum_{(i,j)\in W} (I_1(i,j) \text{ bitwiseXOR } I_2(x+i, y+j))$

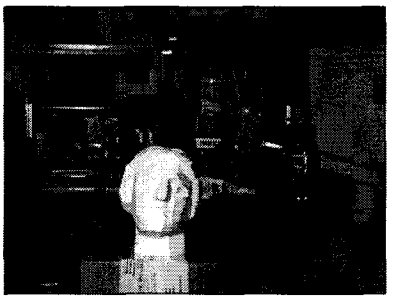
Sum of Absolute Differences (SAD) is one of the simplest of the similarity measures which is calculated by subtracting pixels within a square neighborhood between the reference image I_1 and the target image I_2 followed by the aggregation of absolute differences within the square window, and optimization with the winner-take-all (WTA) strategy [**29**]. If the left and right images exactly match, the resultant will be zero.

In Sum of Squared Differences (SSD), the differences are squared and aggregated within a square window and later optimized by WTA strategy. This measure has a higher computational complexity compared to SAD algorithm as it involves numerous multiplication operations.

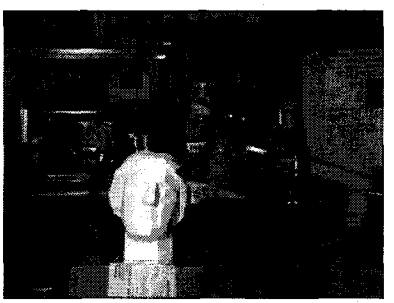
Normalized Cross Correlation is even more complex to both SAD and SSD algorithms as it involves numerous multiplication, division and square root operations.

Sum of Hamming Distances is normally employed for matching census-transformed images by computing bitwise-XOR of the values in left and right images, within a square window. This step is usually followed by a bit-counting operation which results in the final Hamming distance score.

Following figure illustrates the results of SAD, SSD, NCC, and SHD on Tsukuba image from 2001 dataset:

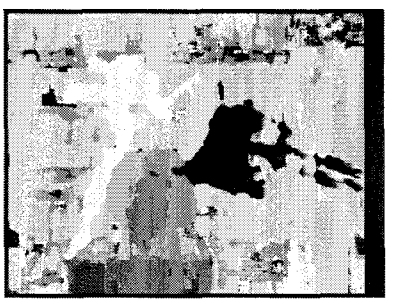


Left Image

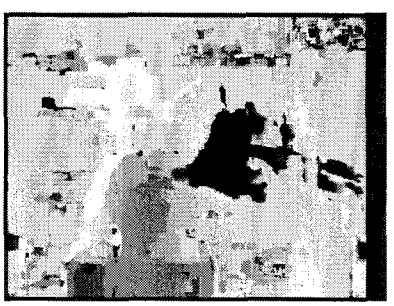


Right Image

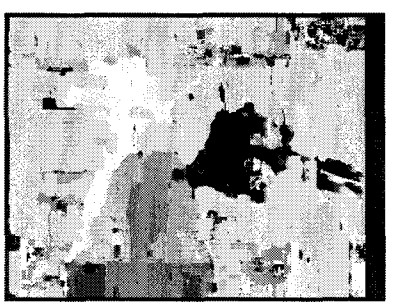
Ground Truth Disparity Map



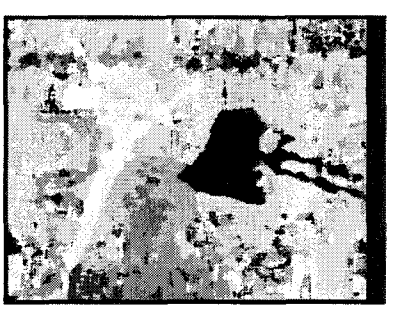
SAD Disparity Map (Window Size: 9x9)



SSD Disparity Map (Window Size: 9x9)



NCC Disparity Map (Window Size: 9x9)



SHD Disparity Map (Window Size: 9x9)

Figure 211. Disparity maps computed by employing SAD, SSD, NCC and SHD on Tsukuba image from 2001 dataset.

The code used to produce SAD, SSD, NCC and SHD scores of left and right images can be found in Appendix A.9-A.12, A.18-A.19.

The work done in [**30**], [**31**] has been extended further, to employ vertical, horizontal and modular recursion schemes for calculating the sum of hamming distance (SHD) scores.

For a given disparity range d, and a square window of size $(2n+1)$x$(2n+1)$, centered at (x,y) in the left census transformed image $I_{L\text{-}CENSUS}$, and $(x+d,y)$ in the right census transformed image $I_{R\text{-}CENSUS}$ (see Figure 22), the sum of hamming distance score $SHD(x,y,d)$ can be given by:

$$SHD(x,y,d) = \sum_{i,j=-n}^{n} I_{L-CENSUS}(x+j,y+i) \otimes I_{R-CENSUS}(x+d+j,y+i) \quad \textbf{(10)}$$

where, $\otimes$ is the bit-wise XOR operator.

If we store the SHD scores associated with all the centre pixels comprising the width of the image (W), for all possible disparity range values ($d_{max}+1$), then the SHD scores associated with all the centre pixels in the subsequent row $SHD(x,y+1,d)$ can be calculated recursively by taking the SHD score $SHD(x,y,d)$ before the vertical window shift, and excluding the contributions of the row above the shifted window, and considering the contributions from the new row within the shifted window (see Figure 22):

$$SHD(x,y+1,d) = SHD(x,y,d) + U(x,y+1,d) \quad \textbf{(11)}$$

where, U(x,$y+1$,d) is called the update term, given by:

$$U(x,y+1,d) = \sum_{j=-n}^{n} I_{L-CENSUS}(x+j,y+n+1) \otimes I_{R-CENSUS}(x+d+j,y+n+1) - \sum_{j=-n}^{n} I_{L-CENSUS}(x+j,y-n) \otimes I_{R-CENSUS}(x+d+j,y-n) \quad \textbf{(12)}$$

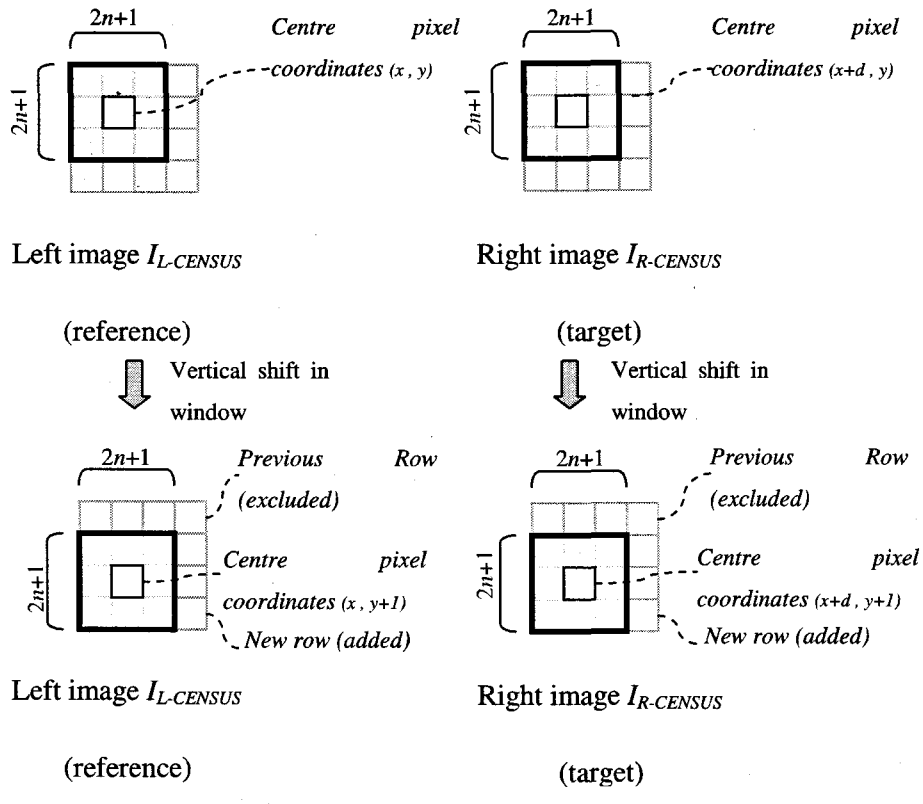


Figure 22. Vertical recursion scheme for correlation based matching. SHD scores associated with window of size 3x3 centered at *(x,y)* in left pre-processed image, and *(x+d,y)* in right pre-processed image, are depicted in light yellow color. SHD scores associated with window centered at *(x,y+1)* in left pre-processed image, and *(x+d,y+1)* in right pre-processed image are depicted in light blue color.

This optimization scheme above is called *vertical recursion.*

Similarly, if we store the update terms for all possible values of disparity (d_{max}+1) associated with the centre pixel to the left of the current centre pixel (see Figure 23a), then it can be shown that the new update terms associated with the current centre pixel $U(x, y+1,d)$ can be recursively calculated from the previous update terms $U(x-1, y+1, d)$ by excluding the contributions of the two (left) corner pixels in previous update terms and considering the contributions of the two (right) corner pixels in the current update terms (see Figure 23b):

$$U(x,y+1,d) = U(x-1,y+1,d) + I_{L-CENSUS}(x+n,y+n+1) \otimes I_{R-CENSUS}(x+d+n,y+n+1)$$
$$+ I_{L-CENSUS}(x+n,y-n) \otimes I_{R-CENSUS}(x+d+n,y-n)$$
$$- I_{L-CENSUS}(x-n-1,y+n+1) \otimes I_{R-CENSUS}(x+d-n-1,y+n+1)$$
$$- I_{L-CENSUS}(x-n-1,y-n) \otimes I_{R-CENSUS}(x+d-n-1,y-n) \quad \textbf{(13)}$$

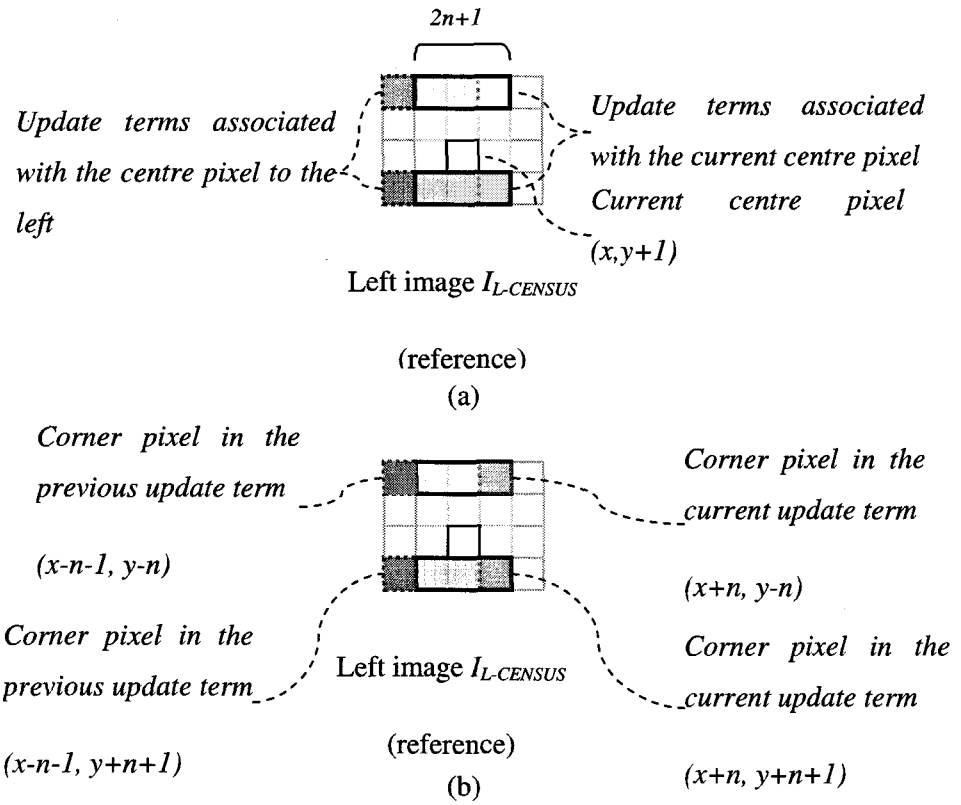


Figure 23. Horizontal recursion scheme for correlation based matching. (a) Update terms associated with the centre pixel to the left are shown with dotted border, whereas update terms associated with the current centre pixel are shown in thick solid border. (b) Corner pixels of the update terms are shown in green color.

This optimization scheme above is called *horizontal recursion*.

From the above, it can be concluded that the run-time of matching algorithm would be independent of window size, which in turn enhances the system flexibility. Another advantage of this approach is the small amount of memory required, which is equal to *WxD* words, where *W* is the image width and *D* is the disparity range. We used the modular recursion as shown in Equations 14-16 . It is based on the observation that contributions associated with the two pixels at the top right corner of correlation window is the same as the contributions of the top left pixels after *2n+1* iterations. Thus, the most recent *2n+1* terms can be temporarily stored within a 2D array *M(x',d)* for each disparity value $d \in [0, d_{max}]$, and can be re-used *2n+1* steps later to reduce the execution time of the algorithm [**30**], [**20**].

$$U(x,y+1,d) = U(x-1,y+!,d) + I_{L-CENSUS}(x+n,y+n+1) \otimes I_{R-CENSUS}(x+d+n,y+n+1) - I_{L-CENSUS}(x+n,y-n) \otimes I_{R-CENSUS}(x+d+n,y-n) - M(x',d) \quad \textbf{(14)}$$

Array *M(x',d)* can be updated modularly as follows:

$$M(x',d) = I_{L-CENSUS}(x-n-1,y+n+1) \otimes I_{R-CENSUS}(x+d-n-1,y+n+1) - I_{L-CENSUS}(x-n-1,y-n) \otimes I_{R-CENSUS}(x+d-n-1,y-n) \quad \textbf{(15)}$$

where,

$$x' = x \bmod (2n+1), d \in [0, d_{max}] \quad \textbf{(16)}$$

For more details on the above, please refer to [**20**], [**30**].

To treat the half-occluded regions in the scene, and enhance the robustness of the algorithm, we have incorporated an efficient implementation of left/right consistency check (LRC) method that produces accurate results under variety of conditions, compared to other occlusion detection methods available [**32**]. LRC check detects majority of the occluded pixels and performs quite well in highly textured scenes. This is accomplished by temporarily storing the SHD values for each row within an array. In fact, we simply use the SHD values produced during the matching of right to left image, for cross-checking results. As a result, no additional memory is required to implement this validation step. For occluded regions, the disparity values are invalidated, and set to 0.

The code used to compute the LRC check can be found in Appendix A.13.

3.4.5 Post-Processing

The post processing step combines the variance map of the target image, variance map of the disparity map and the disparity map itself, to detect and remove erroneous disparities caused by texture-less and depth discontinuous regions in the scene. If the pixel of interest lacks enough texture, and the estimated disparity value is not in agreement with the neighboring pixels (measured by comparing variation of disparity values over a local neighborhood against a threshold), the estimated disparity for that pixel is invalidated and removed from the disparity map (set to zero). This is reasonable for the background and occluded areas, but for the foreground, it may cause holes. The threshold values are determined through manual tuning [**33**]. The variance maps needed for the calculation are produced using three levels of recursion, similar to the matching process [**20**], [**19**].

The code used to post-process the disparity maps can be found in Appendix A.28.

3.5 Achieving Real-time Performance

Optimization includes three major steps: compiler based optimization, system based optimization and assembly level optimization. Compiler based optimization performed to maximize the speed, exploits the architectural features such as pipelining, vectorization, and compiler intrinsic functions (e.g. mult_fr1x16 and add_fr1x16).

System level optimization is achieved by partitioning the memory efficiently and streamlining the data flow. To achieve real-time performance, it is essential to consider the processing speed, data transfer rates and how the memory system handles the data during processing. There is a trade-off between the memory access speed and the available physical size of the memory array. At 20 fps, for two gray scale images sub-sampled to QQVGA resolution, we would have (160x120 pixels/frame)x(1 byte/pixel)x(20frames/sec)x(2 cameras) ~750 KB/sec of raw data throughput into the processor. This helps in estimating the amount of processing that is needed for the stereo vision system to function at 20fps. As mentioned in section 3.3.1, each processing core has access to only 64KB of fast L1 memory.

Since each image frame consists of 18.75KB of data, we can fit maximum of 3 images at the same time, within the L1 memory. To achieve a performance advantage, and leverage the dual-core architecture of the processor, we have implemented a two-staged pipelined programming model, where one core's output is the next core's input [**20**]. Processing task separation is optimized by Core A performing image acquisition, sub-sampling, stereo-rectification, and, Core B performing pre-processing, matching, and post-processing.

In a stereo-vision system, rectification and matching algorithms are the most time-consuming steps. The captured images are transferred into the L3 (external) memory using 2D DMA facility. Double buffering scheme allows Core A to process input images, while DMA transfers the next image frame into the external memory. This reduces the overhead on the processor. Using another 2D DMA operation, the image is sub-sampled and stored within the L1 memory.

During the rectification stage, constants like image indexes and coefficients have to be computed for the back-projection and bi-linear interpolation steps, respectively. These parameters need not be computed on the fly, as they take up a lot of core-clock cycles. Thus, the indexes and coefficients for each of the left and right images can be calculated and stored within the memory as a look-up table, to be used for rectification of the subsequent image pairs. The size of the index table can be computed as follows:

(160x120 pixels/frame)x(2 index values/pixel)x(1byte/index) = 37.5KB

Similarly, the size of the coefficients table can be computed as follows:

(160x120 pixels/frame)x(4 coefficients /pixel)x(2bytes/coefficient) = 150KB

From the above, it can be seen, that the total amount of memory required for the two tables would be 375 KB. This far exceeds the memory space available on-chip, therefore, the lookup tables are stored in the external memory. We use double buffering along with parallel DMA transfers to solve the problem of long delays associated with external memory accesses (see Figure 10), resulting in more than 50% improvement in the execution time of the rectification algorithm. The last step involves transferring the final rectified image into the shared L2 memory, using 2D DMA.

In order to improve the performance of our matching algorithm (implemented on Core B), we have introduced an in-place processing approach to reduce the amount of memory required. In the in-place processing scheme, the source and destination memory locations remain the same. By keeping the processing core's access within the fast L1 memory, the associated performance bottlenecks and degradations caused by slow memory latencies, are alleviated.

We have also implemented the critical loops in assembly language, which allows us to leverage the efficient programming features provided by the BlackFin architecture such as specialized instructions (e.g. bit-counting facility for calculation of Hamming distance metric), utilization of multiple operations per cycle, hardware loop constructs, specialized addressing modes and interlocked instruction pipelines. The following figure shows the profile of the code used in MESVS system:

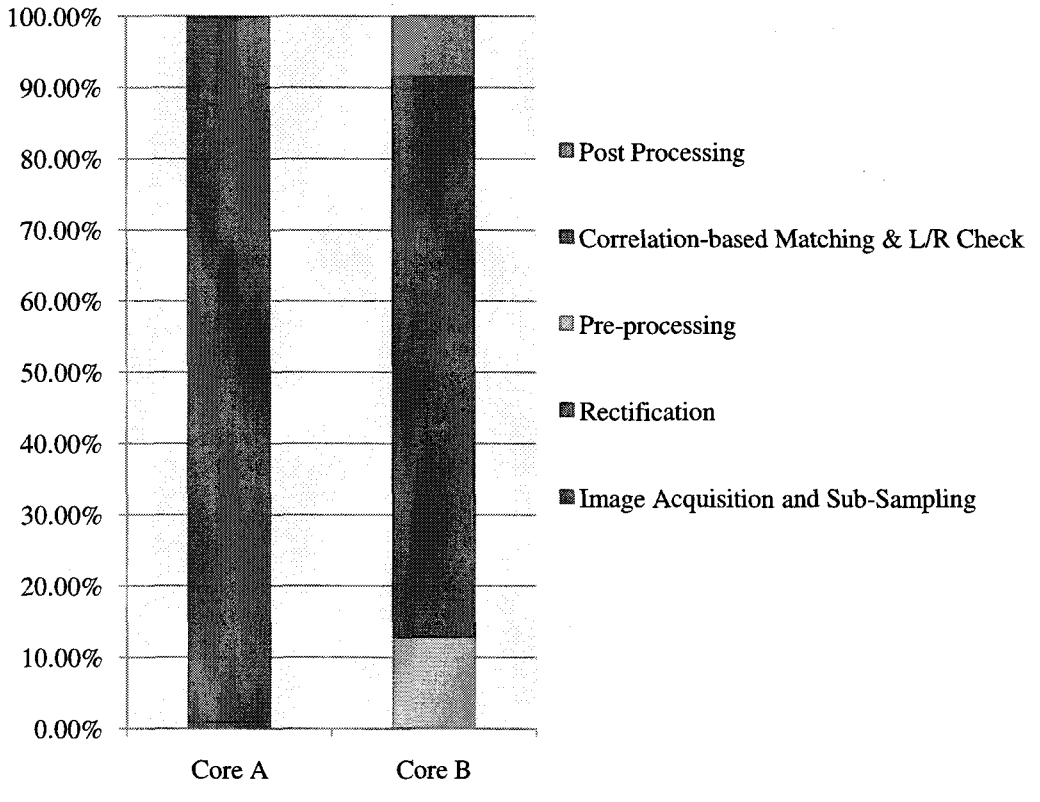


Figure 242. Code profile of MESVS module.

3.6 Quality Metrics

Performance of the stereo vision algorithm can be evaluated by measuring the quality of the computed depth map against the ground-truth disparity map.

3.6.1 Root Mean Squared Error

This measure is computed by the following formula:

$$\boldsymbol{RMS\ Error} = \left(\frac{1}{N}\sum_{(x,y)}|\boldsymbol{d_c}(x,y) - \boldsymbol{d_t}(x,y)|^2\right)^{\frac{1}{2}} \quad \textbf{(17)}$$

where, N is the total number of pixels in an image, d_c is the computed disparity map, and d_t is the ground truth disparity map.

3.6.2 Percentage of Bad Matching Pixels

This measure is computed by the following formula:

$$\%\ \boldsymbol{Bad\ Matching} = \frac{1}{N}\sum_{(x,y)}(|\boldsymbol{d_c}(x,y) - \boldsymbol{d_t}(x,y)| > \boldsymbol{\delta_{thresh}}) \quad \textbf{(18)}$$

where, N is the total number of pixels in an image, d_c is the computed disparity map, and d_t is the ground truth disparity map, and δ_{thresh} is the threshold for evaluating bad matched pixels (usually attains the value of 1.0).

The code used to measure the RMS error and percentage of bad matching pixels can be found in Appendix A.14-A.15.

Stereo vision algorithms typically compute erroneous results when the objects in the scene are fully occluded or half occluded, there are sudden depth changes or discontinuities and when the object has very low texture. To further quantify the performance, maps of the above mentioned regions are constructed from the reference image and ground-truth disparity map. Table 7 shows the definitions of the occluded, depth-discontinuous and texture-less regions.

Table 7. Definitions of occluded, depth-discontinuous and texture-less regions [2].

Occluded regions	Depth-Discontinuous regions	Texture-less regions
Regions that are occluded in the matching image, i.e., where the left-to-right disparity lands at a location with a larger disparity [2].	Regions where neighboring disparities differ by more than a certain gap, dilated by a window of a given width [2].	Regions where the squared horizontal intensity gradient averaged over a square window of a given size is below a given threshold [2].

Chapter 4

Experimental Results

This chapter presents the practical experiments that were performed and the corresponding results obtained using MESVS. It begins with a quantitative comparison of the performance of Rank and Census transforms, followed by the quantitative comparison of the performance of post-processing algorithm. Next the results obtained from the MESVS module for each one of the algorithm stages are presented. Finally an experiment is designed to measure the average power consumption of the system.

4.1 Quantitative comparison of the performance of Rank and Census Transforms

A recent study [**17**] compared the commonly used stereo matching costs such as Birchfield and Tomasi (BT), normalized cross-correlation (NCC), Laplacian of Gaussian (LoG), hierarchical mutual information (HMI), rank and mean filter based costs. It was found, that rank transform appeared to be the best cost for the correlation-based method. Accordingly, the first generation of our system (MESVS-I) relied on rank transform for pre-processing the image pairs [**20**].

It has previously been shown that when census transform is used for pre-processing images, better disparity results are obtained due to lower incorrect matches as compared to rank transform [**26**], [**27**]. However, a comprehensive qualitative and quantitative comparison of rank versus census transform, under radiometric variations such as global and local intensity changes, is lacking in the literature.

In this experiment, we measure the performance of rank and census transforms in the presence of global intensity changes (such as gain, and exposure variances), local intensity changes (such as vignetting, non-Lambertian surfaces, and variable lighting), and image noise.

4.1.1 Simulated Radiometric Variations

The first set of experiments consist of applying artificial radiometric variations to the Middlebury stereo datasets Tsukuba, Venus, Teddy and Cones, as these images were captured under similar lighting conditions [2], [34]. The simulated radiometric changes include: linear global brightness change (gain/scale change), non-linear global brightness change (gamma change), application of vignetting effect, and the introduction of Gaussian noise. The images were then pre-processed using rank (3x3 window size) and census (3x3 window size) transforms. Figure 25 shows the left images of each dataset, followed by the effects of scale change, gamma variation, vignetting, introduction of Gaussian noise, and respective rank and census transforms of left images. The disparity images were then computed by using a correlation window of size 17x17, followed by a left-right consistency check for invalidating occlusions and mismatches. These disparity maps were compared to the ground truth and the average error percentage in non-occluded regions was computed with the error threshold set to 1. We also ignore an area of 8 pixels (half of the correlation window) at the image border.

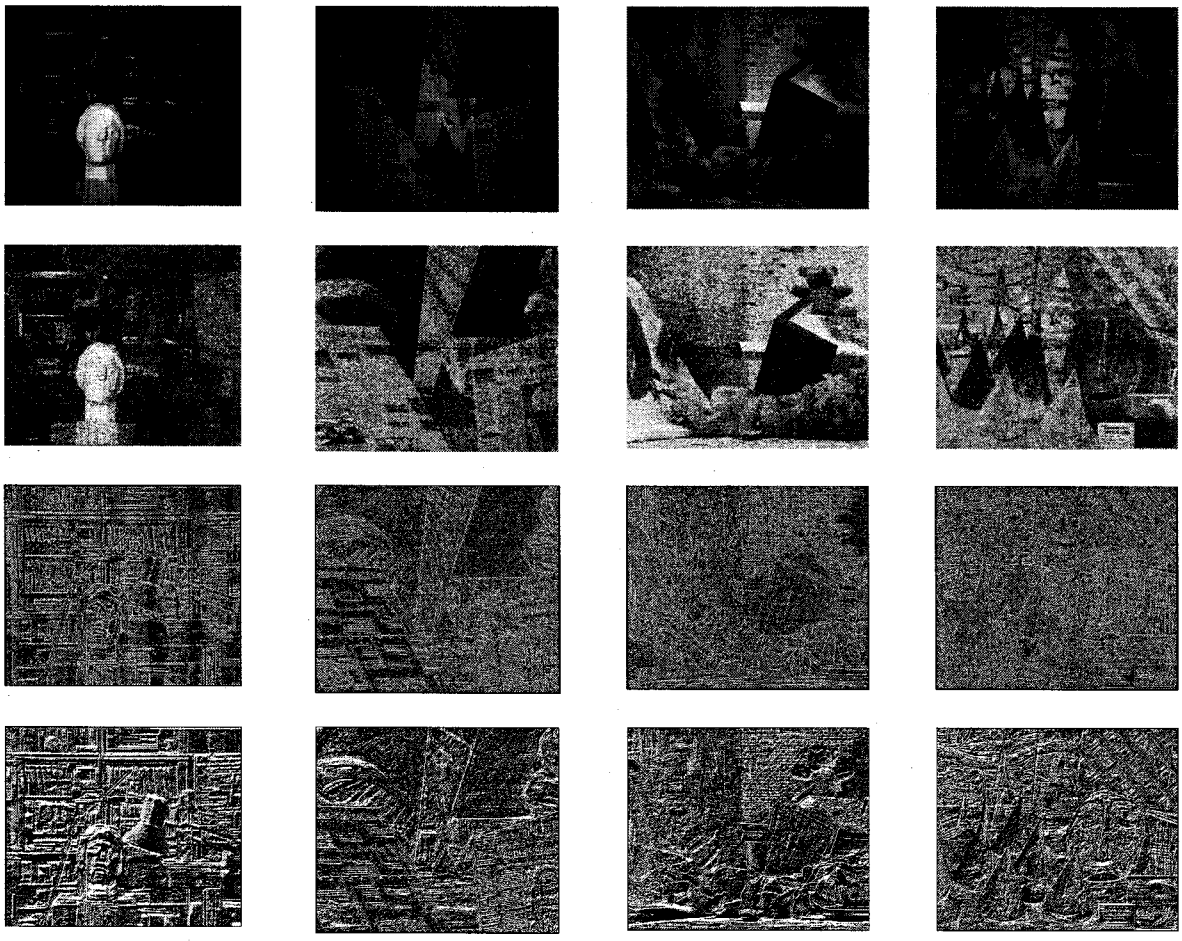
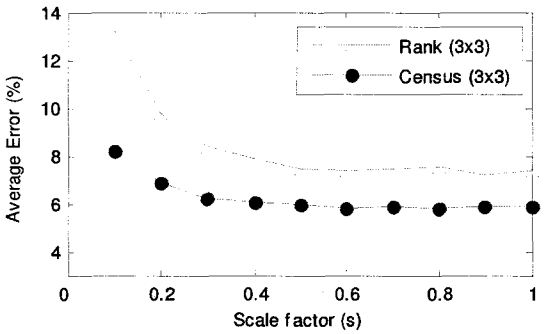
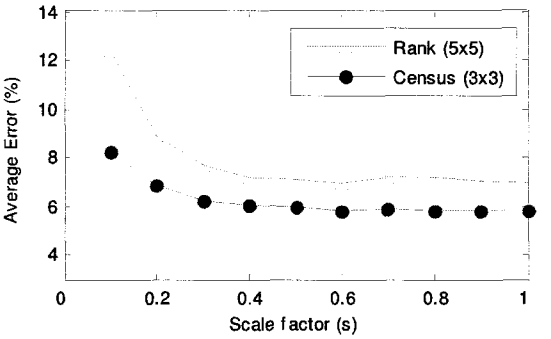


Figure 25. The first row corresponds to the left images of the Tsukuba, Venus, Teddy, and Cones stereo pairs, with subsequent rows displaying various intensity changes such as scale change (s=0.5), gamma change (g=2.5), vignetting effect (sv=0.3) and Gaussian noise (SNR=15dB), as applied to the datasets. The last two rows display the rank (3x3) and census (3x3) transformations of the left images of the datasets.

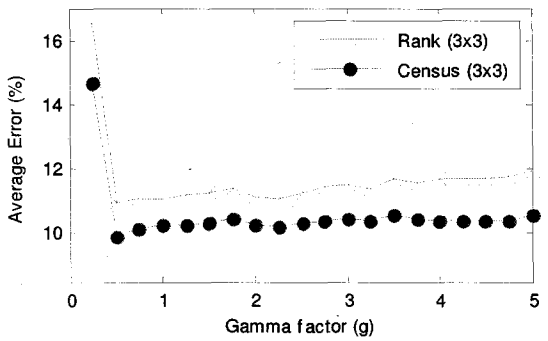
Figure 26 plots the average error percentages in non-occluded regions as a function of the amount of intensity change for both rank and census transforms, within square windows of size 3x3 and 5x5. As can be seen from the experimental results, census transform of size 3x3 outperforms rank of size 3x3 under simulated radiometric variations.



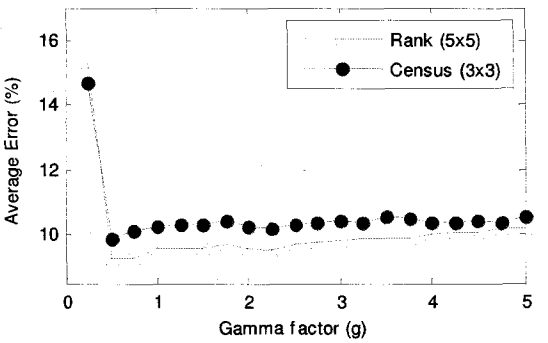
(a) Global scale change-Rank 3x3 vs. Census 3x3



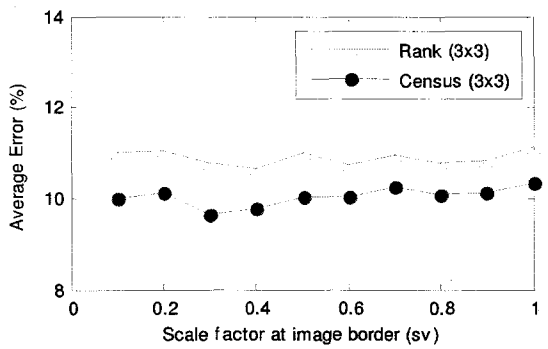
(a) Global scale change-Rank 5x5 vs. Census 3x3



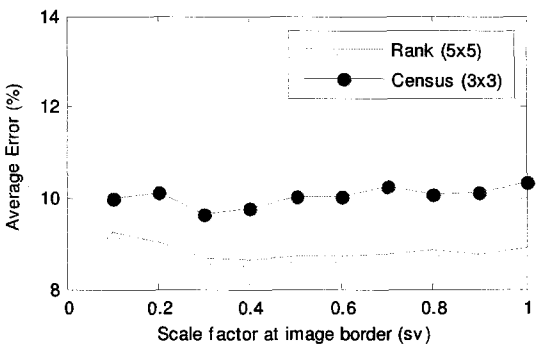
(b) Global gamma change-Rank 3x3 vs. Census 3x3



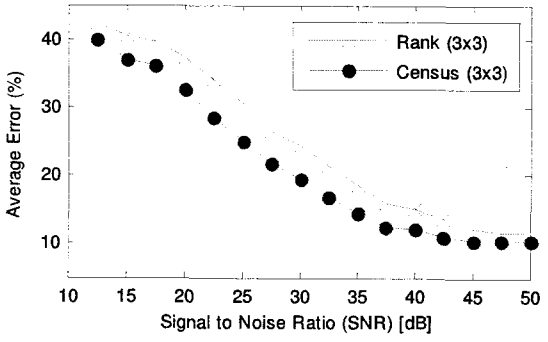
(b) Global gamma change-Rank 5x5 vs. Census 3x3



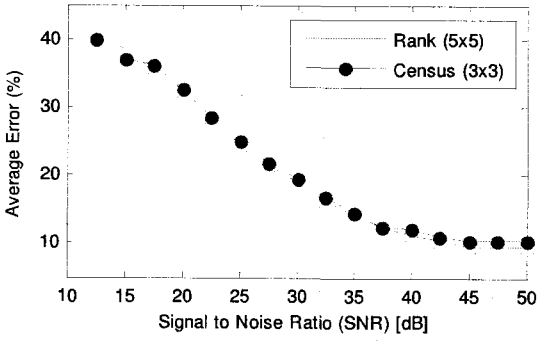
(c) Vignetting-Rank 3x3 vs. Census 3x3



(c) Vignetting-Rank 5x5 vs. Census 3x3



(d) Adding Gaussian noise-Rank 3x3 vs. Census 3x3



(d) Adding Gaussian noise-Rank 5x5 vs. Census 3x3

Figure 26. Effect of applying simulated radiometric changes or noise on the Tsukuba, Venus, Teddy and Cones datasets. The rows correspond to the graphs displaying the average error percentages for Rank (3x3) vs. Census (3x3), and Rank (5x5) vs. Census (3x3) comparisons in the presence of intensity and noise changes.

4.1.2 Real Exposure and Lighting Variations

The second set of experiments consists of measuring the performance of both rank and census transforms under real exposure and lighting variations. In this paper, we used the six datasets (Art, Books, Dolls, Laundry, Moebius, and Reindeer) from Middlebury Stereo [35], [17], with each dataset containing images that were captured under three different exposures and lighting variations, resulting in nine combinatorial pairs of images. Figure 27 shows the left images of these datasets along with their respective ground truths, obtained using the structured lighting technique [34]; while Figure 28 shows the left images of the Books dataset with three different exposures and three different lighting conditions.

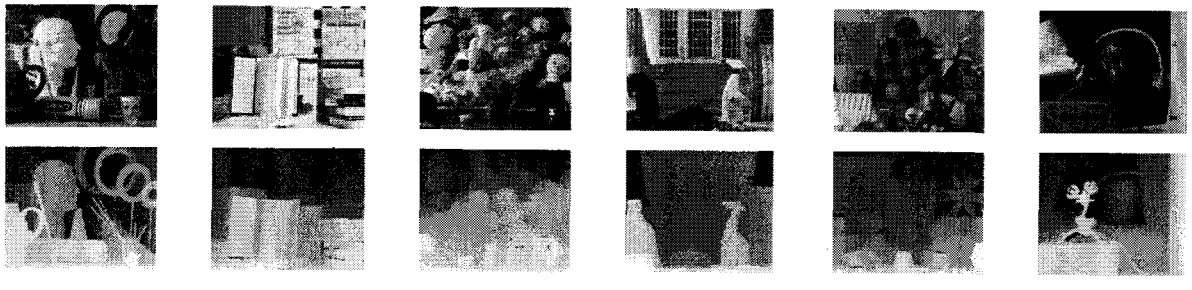


Figure 27. Stereo Datasets (2005) showing test images Art, Books, Dolls, Laundry, Moebius, and Reindeer, along with their ground truth maps.

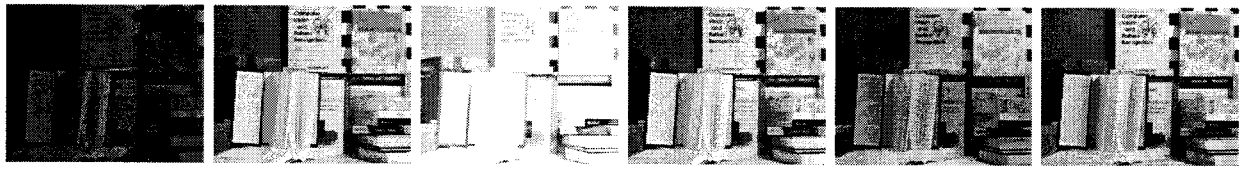
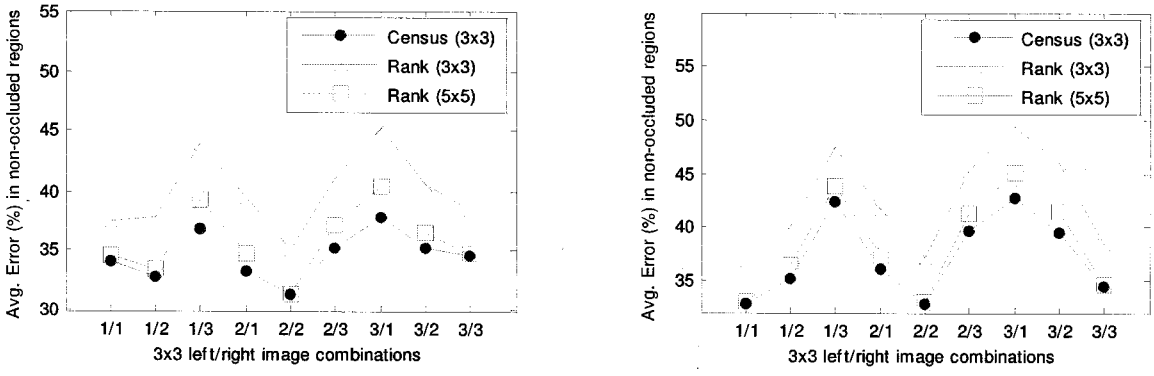


Figure 28. Books dataset displaying left camera images under three different exposures and three different lighting conditions.

We followed the same methodology outlined before to calculate the average error percentages. The resulting graphs are shown in Figure 29.



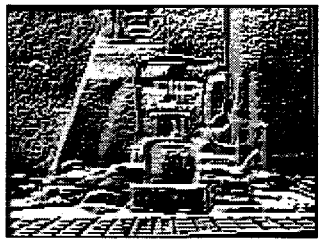
(a) Exposure variation

(b) Lighting variation

Figure 29. Effect of exposure and lighting variations on the Stereo Datasets (2005). The rows correspond to the graphs displaying the average error percentages for Rank (3x3) and Rank (5x5) vs. Census (3x3) comparisons with exposure and lighting variations.

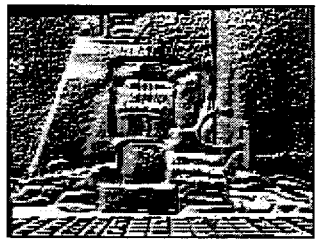
As can be seen from the experimental results above, census transform of size 3x3 outperforms rank of size 3x3 and rank of size 5x5 under real exposure and lighting variations. Accordingly, in the latest generation of the stereo vision system (MESVS-II), we utilize census transform (size 3x3), along with the Hamming distance metric for matching. They are implemented efficiently as a mix of C and assembly code, utilizing BlackFin's special, in-built, single-cycle instructions such as bit counting (Ones), and bit-wise exclusive-or (XOR).

Use of these instructions, combined with the carefully optimized code has allowed us to develop a fast pre-processing stage, which is robust with respect to radiometric variations. Figure 30 displays the census transforms of both left and right images obtained from our module (MESVS-II).



Left pre-processed image

$(I_{L\text{-}C})$



Right pre-processed image

$(I_{R\text{-}C})$

Figure 30. Left and Right pre-processed images (using census transform) belonging to the Mug dataset obtained from MESVS-II.

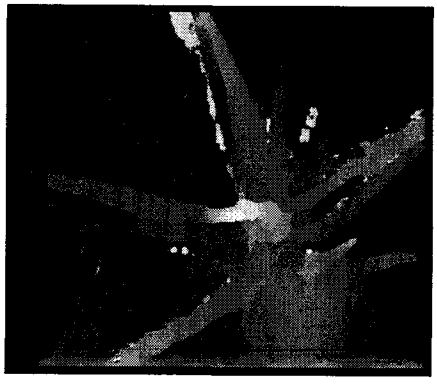
4.2 Quantitative comparison of the performance of post-processing algorithm

Post processing step takes as an input the variance map of the left image, disparity map after LRC check $D_f(x,y)$ and the variance map of the disparity map. The final estimated disparity value for a pixel is invalidated and marked zero whenever its texture $\sigma_I(x,y)$ is less than a threshold T_L and its disparity variation $\sigma_D(x,y)$ is more than the threshold T_H. It will be marked as valid and will be retained otherwise. The values for T_L and T_H are manually tuned for best performance [**33**].

Following equation shows the mathematical representation for the proposed idea [**20**]:

$$D_f(x,y) = \begin{cases} 0, & if \begin{cases} \sigma_I(x,y) \leq T_L \\ \sigma_D(x,y) \geq T_H \end{cases} \\ D(x,y) & otherwise \end{cases} \quad \textbf{(19)}$$

Figure 31 shows the disparity map (left-to-right matching), disparity map after LRC check and the disparity map obtained after post-processing.



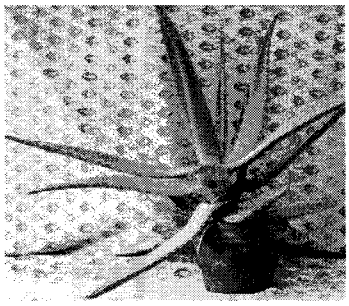
Disparity Map (Left-Right Matching)

Disparity Map after LRC Check

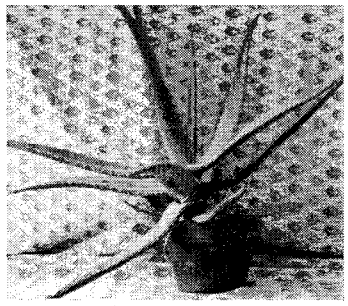
Post processed Disparity Map

Figure 31. Showing the computed disparity map (left-to-right matching), disparity map after LRC Check and Post-processed disparity map of Aloe image from 2006 dataset.

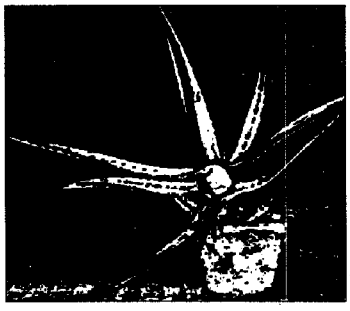
To quantify the results of the post-processing algorithm in various challenging regions, 2006 dataset from Middlebury Stereo website was chosen as it contains a variety of real-world images **[35]**, **[17]**. Figure 32 shows left and right images of Aloe dataset, ground-truth disparity map, along with the obtained texture-less, occluded and depth-discontinuous maps.



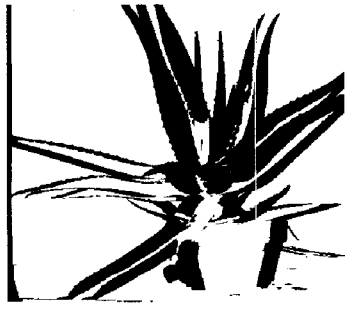
Left Image



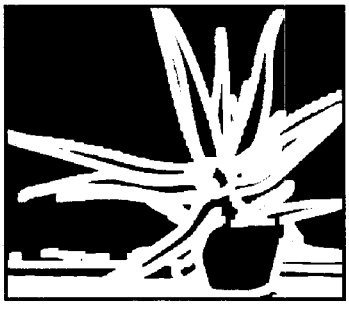
Right Image



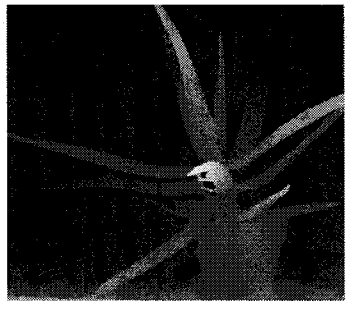
Texture-less Map



Occlusion Map



Depth-Discontinuity Map

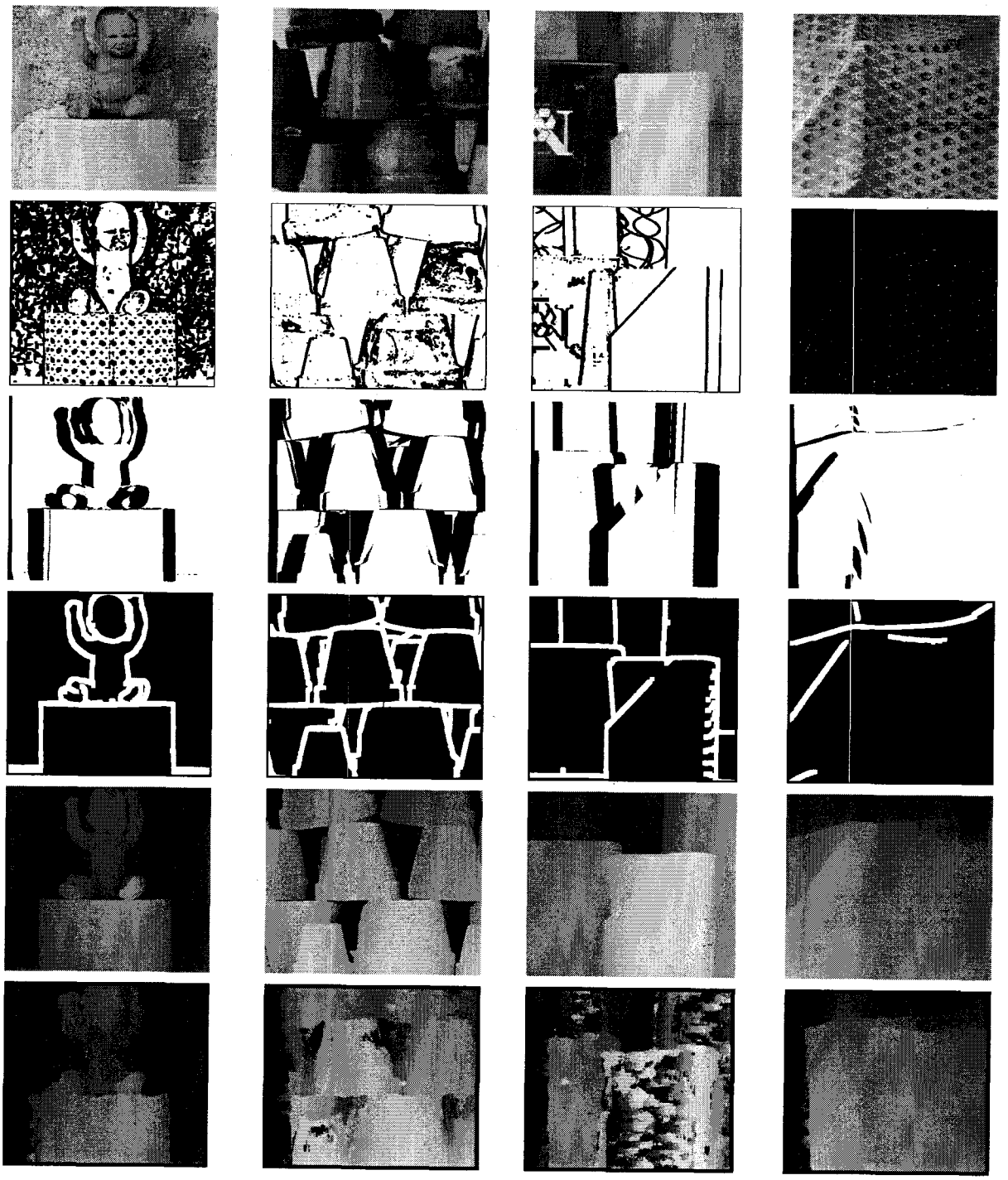


Ground-Truth Disparity Map

Figure 32. Shown from left to right, left image, right image, texture-less map, occlusion map, depth-discontinuous map and ground-truth disparity map of Aloe image from 2006 dataset.

Figure 33 shows some of the challenging images from the 2006 dataset. Baby 1 dataset contains a highly textured background and a relatively low textured foreground. Flowerpots dataset contains sudden depth changes and large number of occluded regions. Plastic dataset poses a tremendous challenge as it contains mostly low-textured regions, and a lot of occluded areas. Cloth1 dataset is heavily texture, and has very low number of pixels that fall in occluded and depth –discontinuous regions. Also shown in the

figure are the ground-truth disparity maps, computed disparity map, disparity map after LRC check and disparity map after post-processing.



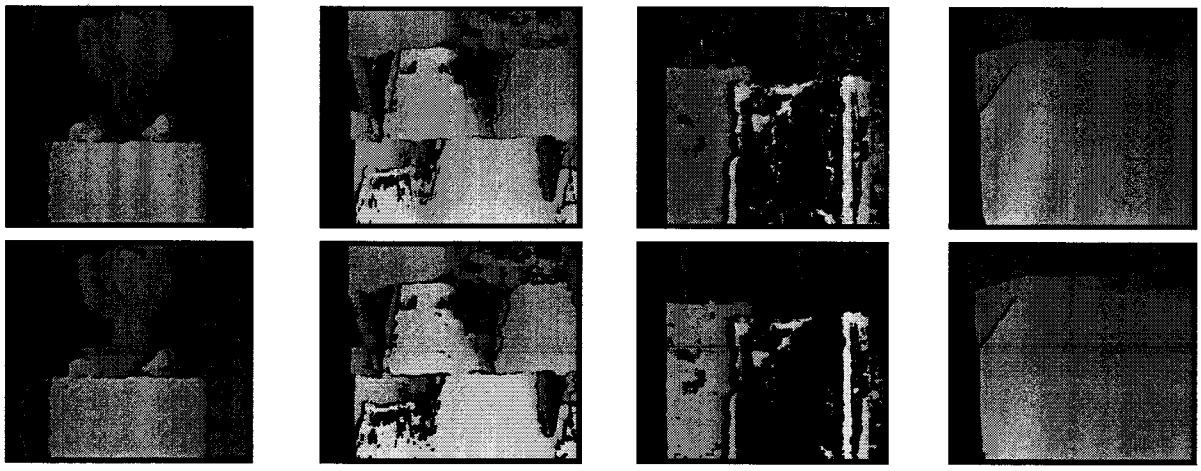
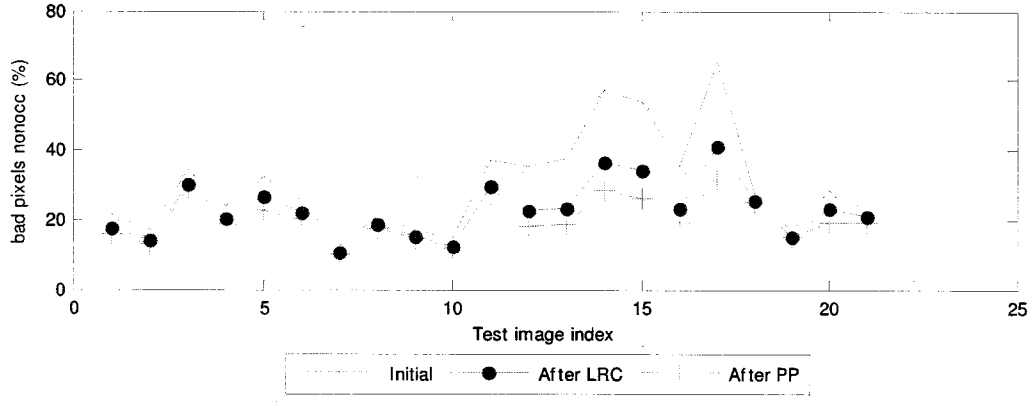
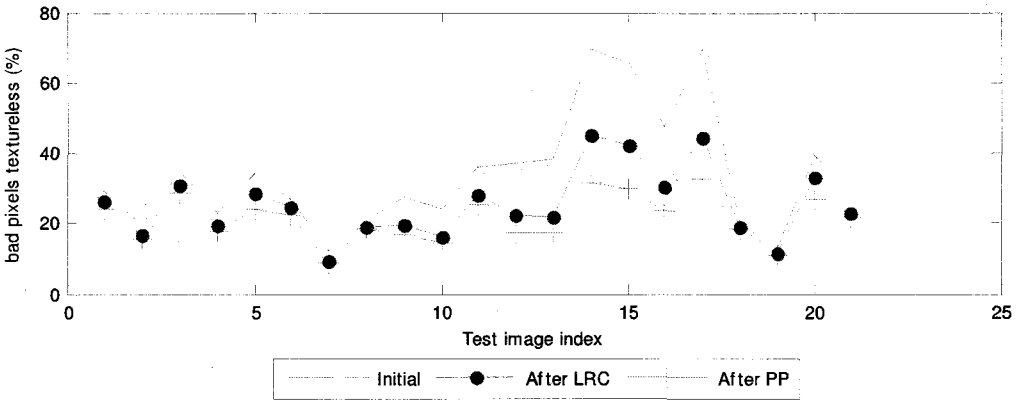


Figure 33. Challenging images. From top to bottom: left image, texture-less map, occlusion map, depth-discontinuous map, ground-truth disparity map, computed disparity map (left-to-right matching), disparity map after LRC, and final post-processed disparity map. From left to right, images Baby1, Flowerpots, Plastic, and Cloth1 from 2006 dataset.

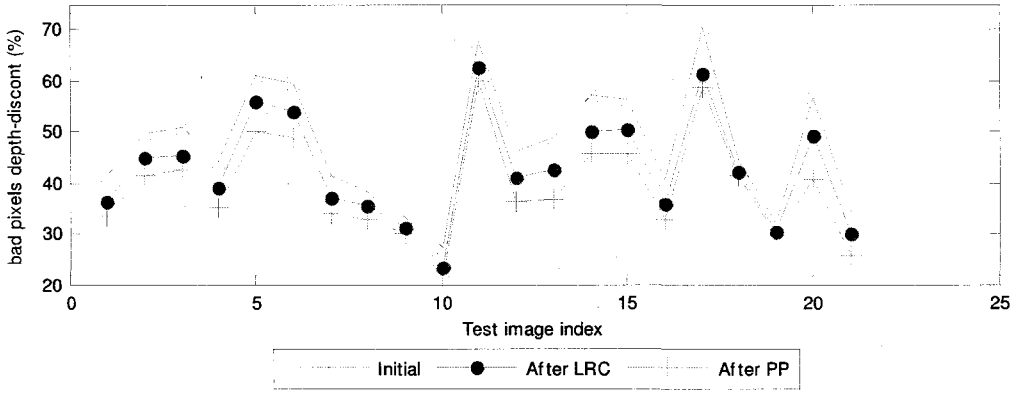
In each one of the non-occluded, texture-less and depth discontinuous regions, percentage of bad matching pixels for each one of the 21 images, after initial disparity computation (Initial), after LRC Check and after post-processing was computed (see Figure 34). If the post-processing algorithm results in lower number of bad matching pixels, then we can safely say that the algorithm works by eliminating bad matched pixels from the final depth map.



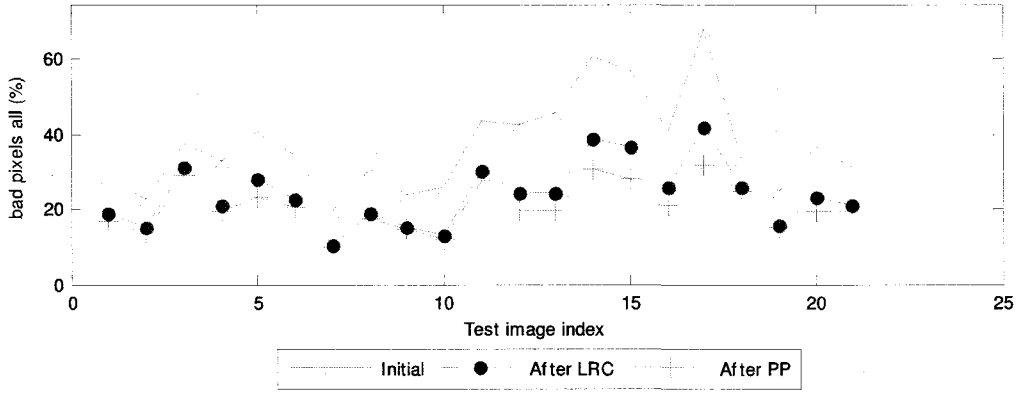
(a) Percentage of Bad matching pixels in non-occluded regions.



(a) Percentage of Bad matching pixels in textureless regions.



(a) Percentage of Bad matching pixels in depth-discontinuous regions.



(a) Percentage of Bad matching pixels in all regions.

Figure 34. Percentage of bad matching pixels after initial disparity computation, after LRC check and after Post-processing for non-occluded, texture-less, depth-discontinuous and all regions (top to bottom).

Table shows the mean, median and mode values of the percentage reduction in bad matching pixels obtained after LRC check and post-processing. As can be seen, for all regions, post processing algorithm has an average percentage reduction of bad matched pixels, or in other words average percentage improvement of 13.61%, in non-occluded, texture-less and depth-discontinuous regions the percentage improvement is 13.24%, 15.91% and 8.37% respectively.

Table 8. Percentage reduction in bad matching pixels after LRC check and after post-processing.

	Percentage of Bad Matching pixels	**After Initial Disparity Computation**	**After LRC Check**	**Percentage reduction**	**After Post-processing**	**Percentage reduction**
All regions	Mean (%)	37.30	24.02	35.60	20.75	**13.61**
	Median (%)	34.36	23.54	31.49	19.61	16.69
	Mode (%)	19.74	10.65	46.05	10.13	4.88
Non-occluded regions	Mean (%)	30.29	23.04	23.93	19.99	**13.24**
	Median (%)	27.12	22.57	16.78	19.11	15.33
	Mode (%)	12.38	10.53	14.94	10.07	4.37
Texture-less regions	Mean (%)	33.99	25.33	25.48	21.30	**15.91**
	Median (%)	29.29	22.83	22.05	21.60	5.39
	Mode (%)	11.95	9.285	22.30	8.838	4.81
Depth-discontinuous regions	Mean (%)	47.72	42.77	10.37	39.19	**8.37**
	Median (%)	46.11	42.36	8.13	36.79	13.15
	Mode (%)	27.12	23.27	14.20	21.32	8.38

4.3 Overview of Stereo-matching engine

The results produced by our stereo matching engine are shown in Figure 35. The results presented are obtained with the window size of 3x3 for census transform, and 15x15 for the correlation window. The disparity range is set to 30 with horopter of 5 to 35. Most of the erroneous matches associated with half-occluded regions of the scene (shown in black) are eliminated by LRC algorithm. Qualitative assessment of the post-processing step can be performed by observing the results in of the post-processing algorithm in the figure. As can be seen, the shape of the object is clearly defined, especially around the curvature of the upper portion. It can also be seen that our post-processing algorithm significantly alleviates the foreground fattening problem associated with correlation-based matching methods.

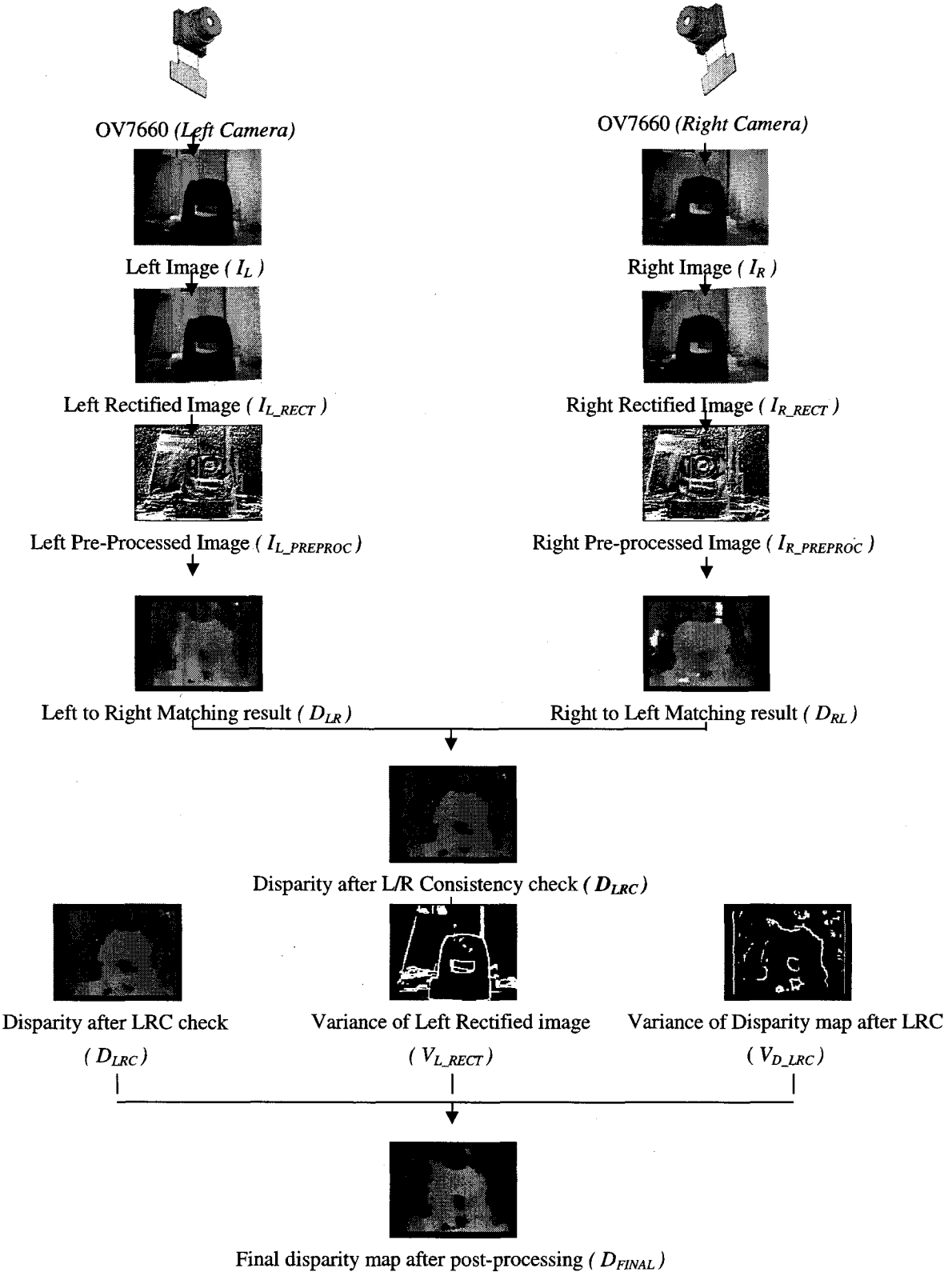


Figure 35. Results of Stereo Vision Engine using another camera as the object of interest.

4.4 Power profile

One of the most important criteria for evaluating the design of an embedded media processing system is its power consumption. This goes without saying for applications that require mobility, but it is also critical for tethered systems, as it has direct financial consequences in terms of energy consumption, number of features that can be added, density of components, packaging, and the overall life-span of parts. For systems that operate at high speeds, and consume a lot of power, heat dissipation, and thermal management becomes a significant challenge. Most often, this necessitates the usage of active cooling by fans to provide airflow for heat removal, adding to the power consumption, component costs and noise.

Passive power management typically is a coordinated effort that involves careful selection of individual parts, optimum component layout and routing schemes, intelligent voltage regulation, separating power domains, implementing power management modes, dynamically changing frequency and voltage, and optimization of software algorithms.

As the next generation processors move from 90nm process to 45nm process and further down the size ladder, the thinner isolation layers lead to high static power dissipation caused by transistor leakage currents during the quiescent state i.e. when the processor is idle. Static power consumption is a constant of the selected processor, which can be reduced by lowering the voltage applied to it, or by lowering the operating temperature. When the processor is not idle, there is variable/dynamic power dissipation P_{dyn} caused by active currents due to charge-discharge cycles of load capacitances at high switching frequencies. It is directly proportional to operating frequency f and square of the processor's core supply voltage V_{DDINT} [**36**], [**22**]:

$$P_{dyn} = KV_{DDINT}^2 Xf \qquad \textbf{(20)}$$

where, K is a constant of load capacitance.

As can be seen from equation (20), lowering the frequency would result in a linear reduction, whereas lowering the applied core-clock voltage results in an exponential decrease in dynamic power consumption. Keeping the voltage constant, and lowering just the frequency will not have the desired effect, as it will take longer for the code to run. For our system, we first lower the operating voltage, and only when no further decrease in voltage is possible, do we go for reduction in frequency.

In the MESVS-II system, apart from the dynamic management of frequency and voltage control, we have also incorporated several other power management features such as flexible operating power modes (such as full-on, active, sleep, deep-sleep and hibernate), separation of power domains (processor's core voltage rail separated from I/O supply rail and phased-locked-loop), and intelligent voltage regulation. In addition, we have also carefully tuned the algorithms to reduce the execution time using the Visual DSP++ tool suite's built in statistical profiler. The full-on, average power consumption of the MESVS-II module is around 2.3W (700mA @ 3.3V) as shown in figures 37 and 38 **[20]**, **[19]**.

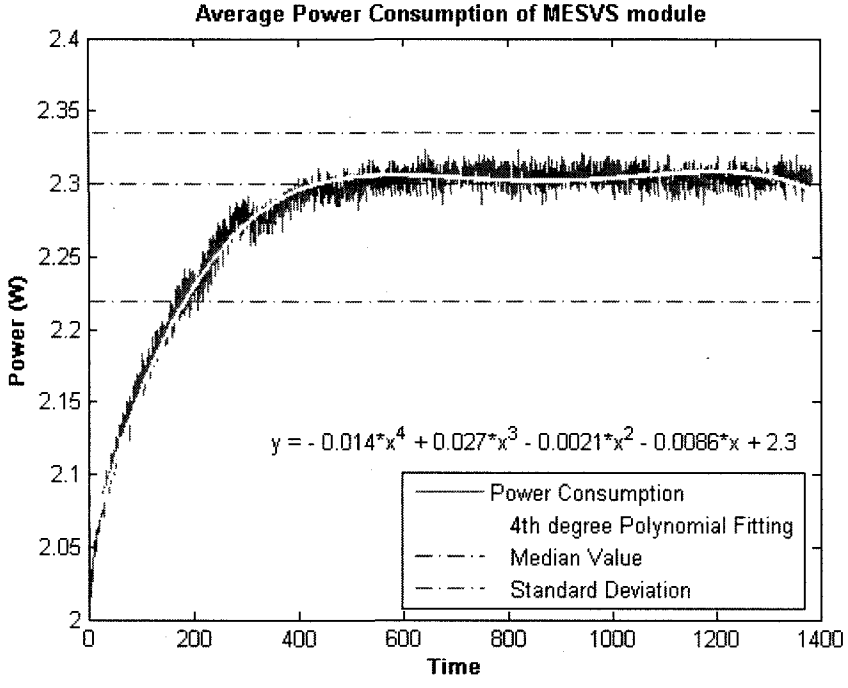


Figure 37. Average power consumption of MESVS Module

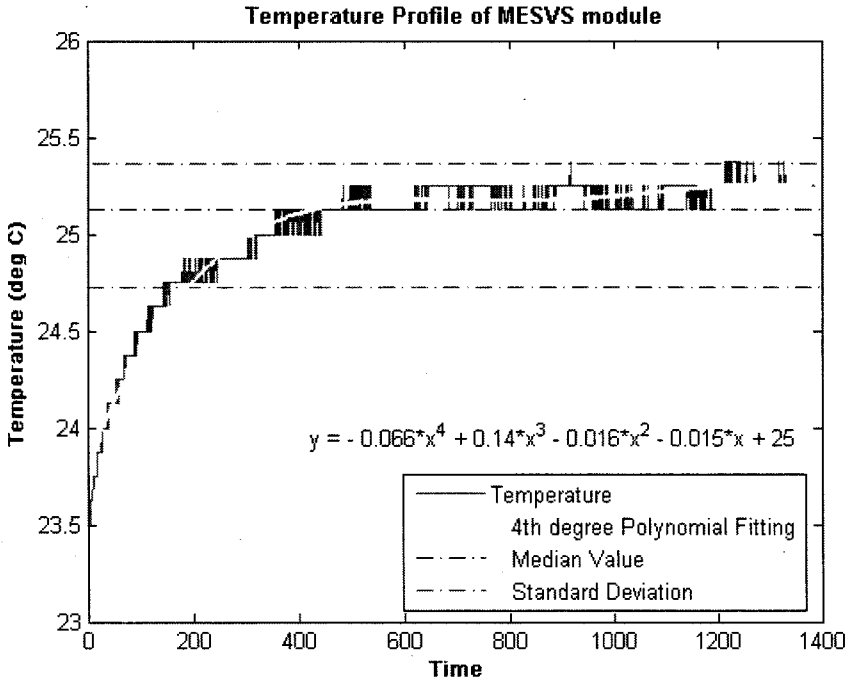


Figure 38. Temperature profile of MESVS module

Chapter 5

Conclusions

5.1 Summary of contributions

The goal of this work was to develop a fully integrated, small baseline (<=3cm), miniaturized embedded stereo vision system which fits into a tiny package of 5x5cm and consumes very low power (700mA@3.3V). The low cost (<=$600) system consists of two small profile CMOS cameras (image resolution QQVGA-160x120), and a power efficient, dual-core embedded media processor, running at 600MHz per core. The stereo-matching engine performs sub-sampling, rectification, pre-processing using rank transform, correlation-based SHD (Sum of Hamming Distance) matching using three levels of recursion, L/R consistency check and post-processing. A novel post processing algorithm has been proposed that removes outliers due to low-texture regions and depth-discontinuities by combining the contributions from the variance map of the rectified image, disparity map, and the variance map of the disparity map. A quantitative performance of the post processing algorithm has been presented which shows that for all regions, post processing algorithm has an average percentage reduction of bad matched pixels, or in other words average percentage improvement of 13.61%, in non-occluded, texture-less and depth-discontinuous regions the percentage improvement is 13.24%, 15.91% and 8.37% respectively.

To further enhance the performance of the system, compiler based optimization, system based optimization and assembly level optimization has been performed. A two staged pipelined-processing scheme has been implemented, that takes advantage of the dual-core architecture of the embedded processor, thereby achieving a processing speed of around 10fps for disparity maps in MESVS-I system and 20fps in MESVS-II system.

To enhance the robustness of the pre-processing stage to radiometric variations, census transform (size 3x3) has been employed, which has been shown to out-perform rank (size 3x3 and 5x5). High quality depth map results obtained have also been presented.

Some of the applications include miniaturized mobile robotics, drowsy driver detection, and 3D object tracking etc.

5. 2 Future Work

Some of the areas where the work can be extended in future include

- Improving the results in challenging areas like texture-less, occluded and depth discontinuous regions.
- Increase the frame rate and image resolution by further optimization of algorithms and software algorithms.
- Investigate the feasibility of incorporating a co-processors (FPGA/DSP)
- Implement a global algorithm on the system.

Appendix A

Software Source Code

A.1 Rectify Left and Right Images

```
% ************************************************************************
% Title: Function-Rectify left and right images
% Author: Siddhant Ahuja
% Created: September 2008
% Copyright Siddhant Ahuja, 2008
% Inputs: Input Left Image (var: leftImage), Input Right Image (var:
rightImage),
% Focal Length of Left Camera (var:
% fc_left), Principal point of Left Camera (var: cc_left), Skewness in the
% left camera (var: alpha_c_left), Distortion in the left camera (var:
% kc_left), Focal Length of Right Camera (var: fc_right), Principal point
% of Right Camera (cc_right), Skewness in the Right camera (var:
% alpha_c_right), Distortion in the right camera (var: kc_right), Rotation
% Vector (var: om), Translation Vector (var: T), Number of rows (var: nx),
Number of
% columns (var: ny)
% Outputs: Rectified Image (var: rectifiedImg),
% Time taken (var: timeTaken)
% Example Usage of Function: [rectifiedImg,
timeTaken]=funcRectify('left_sample_5.bmp', 'right_sample_5.bmp', [
% 755.06557   755.16795 ], [ 334.24409   251.98013 ], [ 0.00000 ], [
% 0.05080   -0.21590   -0.00997   0.01058  0.00000 ], [ 758.73630
% 759.13330 ], [ 317.44101   230.19193 ], [ 0.00000 ], [ 0.01704   -0.21345
% -0.00647   0.00525  0.00000 ], [ -0.00804   0.00752  0.03008 ], [
% -29.85007;   -0.61101;  -0.15665 ], 640, 480);
% ************************************************************************
function [leftRectifiedImg,
rightRectifiedImg,timeTaken]=funcRectify(leftImage,rightImage,fc_left,cc_left
,alpha_c_left,kc_left,fc_right,cc_right,alpha_c_right,kc_right,om,T,nx,ny);
%% set the rectification parameters of hybrid approach manually
% INTRINSIC PARAMETERS
% Focal Length of Left Camera
% fc_left = [ 755.06557   755.16795 ];
% Principal point of Left Camera
% cc_left = [ 334.24409   251.98013 ];
% Skewness in the left camera
% alpha_c_left = [ 0.00000 ];
% Distortion in the left camera
% kc_left = [ 0.05080   -0.21590   -0.00997   0.01058  0.00000 ];
% Focal Length of Right Camera
% fc_right = [ 758.73630   759.13330 ];
% Principal point of Right Camera
% cc_right = [ 317.44101   230.19193 ];
```

```
% Skewness in the Right camera
% alpha_c_right = [ 0.00000 ];
% Distortion in the right camera
% kc_right = [ 0.01704   -0.21345   -0.00647   0.00525  0.00000 ];
% EXTRINSIC PARAMETERS
% Rotation Vector
% om = [ -0.00804   0.00752  0.03008 ];
% Translation Vector
% T = [ -29.85007;   -0.61101;  -0.15665 ];
% Image width
% nx = 640;
% Image height
% ny = 480;
% Read LeftImage
leftImage = double(rgb2gray(imread(leftImage)));
% Read RightImage
rightImage = double(rgb2gray(imread(rightImage)));
% Find how many rows and columns are there in the left image.
[nr,nc] = size(leftImage);
% Determine scale factor for rectification
scale=floor(nx/nc);
nx=nr; ny=nc;
% Start Timer to measure execution time
tic
%% calculate constants
R = rodrigues(om);
% Bring the 2 cameras in the same orientation by rotating them "minimally":
r_r = rodrigues(-om/2);
r_l = r_r';
t = r_r * T;
% Rotate both cameras so as to bring the translation vector in alignment with
the (1;0;0) axis:
if abs(t(1)) > abs(t(2)),
    type_stereo = 0;
    uu = [1;0;0]; % Horizontal epipolar lines
else
    type_stereo = 1;
    uu = [0;1;0]; % Vertical epipolar lines
end;
if dot(uu,t)<0,
    uu = -uu; % Swtich side of the vector
end;
ww = cross(t,uu);
ww = ww/norm(ww);
ww = acos(abs(dot(t,uu))/(norm(t)*norm(uu)))*ww;
R2 = rodrigues(ww);
% Global rotations to be applied to both views:
R_R = R2 * r_r;
R_L = R2 * r_l;
% Computation of the *new* intrinsic parameters for both left and right
cameras:
% Vertical focal length *MUST* be the same for both images (here, we are
trying to find a focal length that retains as much information contained in
the original distorted images):
if kc_left(1) < 0,
    fc_y_left_new    =    fc_left(2)    *    (1    +    kc_left(1)*(nx^2    +
ny^2)/(4*fc_left(2)^2));
```

```
else
    fc_y_left_new = fc_left(2);
end;
if kc_right(1) < 0,
    fc_y_right_new    =    fc_right(2)    *    (1    +    kc_right(1)*(nx^2    +
ny^2)/(4*fc_right(2)^2));
else
    fc_y_right_new = fc_right(2);
end;
fc_y_new = min(fc_y_left_new,fc_y_right_new);
% For simplicity, let's pick the same value for the horizontal focal length
as the vertical focal length (resulting into square pixels):
fc_left_new = round([fc_y_new;fc_y_new]);
fc_right_new = round([fc_y_new;fc_y_new]);
% Select the new principal points to maximize the visible area in the
rectified images
cc_left_new = [(nx-1)/2;(ny-1)/2] - mean(project_points2([normalize_pixel([0
nx-1 nx-1 0; 0 0 ny-1 ny-1],fc_left,cc_left,kc_left,alpha_c_left);[1 1 1
1]],rodrigues(R_L),zeros(3,1),fc_left_new,[0;0],zeros(5,1),0),2);
cc_right_new = [(nx-1)/2;(ny-1)/2] - mean(project_points2([normalize_pixel([0
nx-1 nx-1 0; 0 0 ny-1 ny-1],fc_right,cc_right,kc_right,alpha_c_right);[1 1 1
1]],rodrigues(R_R),zeros(3,1),fc_right_new,[0;0],zeros(5,1),0),2);
% For simplivity, set the principal points for both cameras to be the average
of the two principal points.
if ~type_stereo,
    %-- Horizontal stereo
    cc_y_new = (cc_left_new(2) + cc_right_new(2))/2;
    cc_left_new = [cc_left_new(1);cc_y_new];
    cc_right_new = [cc_right_new(1);cc_y_new];
else
    %-- Vertical stereo
    cc_x_new = (cc_left_new(1) + cc_right_new(1))/2;
    cc_left_new = [cc_x_new;cc_left_new(2)];
    cc_right_new = [cc_x_new;cc_right_new(2)];
end;
% Of course, we do not want any skew or distortion after rectification:
alpha_c_left_new = 0;
alpha_c_right_new = 0;
kc_left_new = zeros(5,1);
kc_right_new = zeros(5,1);
% The resulting left and right camera matrices:
KK_left_new       =       [fc_left_new(1)       fc_left_new(1)*alpha_c_left_new
cc_left_new(1);0 fc_left_new(2) cc_left_new(2); 0 0 1];
KK_right_new       =       [fc_right_new(1)       fc_right_new(1)*alpha_c_right
cc_right_new(1);0 fc_right_new(2) cc_right_new(2); 0 0 1];
% Apply scale factor
KK_left_new(1,1) = KK_left_new(1,1) / scale;
KK_left_new(1,3) = KK_left_new(1,3) / scale;
KK_left_new(2,2) = KK_left_new(2,2) / scale;
KK_left_new(2,3) = KK_left_new(2,3) / scale;
KK_right_new(1,1) = KK_right_new(1,1) / scale;
KK_right_new(1,3) = KK_right_new(1,3) / scale;
KK_right_new(2,2) = KK_right_new(2,2) / scale;
KK_right_new(2,3) = KK_right_new(2,3) / scale;
fc_left = fc_left / scale;
cc_left = cc_left / scale;
fc_right = fc_right / scale;
```

```
cc_right = cc_right / scale;
inv_KK_left_new = inv(KK_left_new);
inv_KK_right_new = inv(KK_right_new);
%% Apply Rectification to left image
leftRectifiedImg = 255*ones(nx,ny);
for (i=1:nx)
    for (j=1:ny)
        rays = inv_KK_left_new*[j-1; i-1; 1];
        rays2 = R_L'*rays;
        x = [rays2(1,1)./rays2(3,1);rays2(2,1)./rays2(3,1)];
        %% apply distortion
        k = kc_left;
        r2 = x(1,1).^2 + x(2,1).^2;
        r4 = r2.^2;
        r6 = r2.^3;
        % Radial distortion:
        cdist = 1 + k(1) * r2 + k(2) * r4 + k(5) * r6;
        xd(1,1) = x(1,1) * cdist;
        if ((xd(1,1) < -1) | (xd(1,1) >= 1))
           a = 1
        end
        xd(2,1) = x(2,1) * cdist;
        if ((xd(2,1) < -1) | (xd(2,1) >= 1))
           a = 1
        end
        % Tangential distortion:
        a1 = 2.*x(1,1).*x(2,1);
        a2 = r2 + 2*x(1,1).^2;
        a3 = r2 + 2*x(2,1).^2;
        delta_x_1 = k(3)*a1 + k(4)*a2;
        delta_x_2 = k(3) * a3 + k(4)*a1;
        xd(1,1) = xd(1,1) + delta_x_1;
        xd(2,1) = xd(2,1) + delta_x_2;
        %% bilinear interpolation stage
        f = fc_left;
        c = cc_left;
        px2 = f(1) * xd(1,1) + c(1);
        py2 = f(2) * xd(2,1) + c(2);
        % interpolate between the closest pixels
        px_0 = floor(px2);
        py_0 = floor(py2);
        if ((px_0 >= 0) & (px_0 <= (ny-2)) & (py_0 >= 0) & (py_0 <= (nx-2)))
            alpha_x = px2 - px_0;
            alpha_y = py2 - py_0;
            a1 = (1 - alpha_y).*(1 - alpha_x);
            a2 = (1 - alpha_y).*alpha_x;
            a3 = alpha_y .* (1 - alpha_x);
            a4 = alpha_y .* alpha_x;
            value = a1 * leftImage(py_0+1, px_0+1) + a2 * leftImage(py_0+1,
px_0+2) + a3 * leftImage(py_0+2, px_0+1) + a4 * leftImage(py_0+2, px_0+2);
            leftRectifiedImg(i,j) = value;
        end
    end
end
%% Apply Rectification to right image
rightRectifiedImg = 255*ones(nx,ny);
for (i=1:nx)
```

```
    for (j=1:ny)
        rays = inv_KK_right_new*[j-1; i-1; 1];
        rays2 = R_R'*rays;
        x = [rays2(1,1)./rays2(3,1);rays2(2,1)./rays2(3,1)];
        %% apply distortion
        k = kc_right;
        r2 = x(1,1).^2 + x(2,1).^2;
        r4 = r2.^2;
        r6 = r2.^3;
        % Radial distortion:
        cdist = 1 + k(1) * r2 + k(2) * r4 + k(5) * r6;
        xd(1,1) = x(1,1) * cdist;
        xd(2,1) = x(2,1) * cdist;
        %% bilinear interpolation stage
        f = fc_right;
        c = cc_right;
        px2 = f(1) * xd(1,1) + c(1);
        py2 = f(2) * xd(2,1) + c(2);
        % interpolate between the closest pixels
        px_0 = floor(px2);
        py_0 = floor(py2);
        if ((px_0 >= 0) & (px_0 <= (nc-2)) & (py_0 >= 0) & (py_0 <= (nr-2)))
            alpha_x = px2 - px_0;
            alpha_y = py2 - py_0;
            a1 = (1 - alpha_y).*(1 - alpha_x);
            a2 = (1 - alpha_y).*alpha_x;
            a3 = alpha_y .* (1 - alpha_x);
            a4 = alpha_y .* alpha_x;
            rightRectifiedImg(i,j) = a1 * rightImage(py_0+1, px_0+1) + a2 *
rightImage(py_0+1, px_0+2) + a3 * rightImage(py_0+2, px_0+1) + a4 *
rightImage(py_0+2, px_0+2);
        end
    end
end
% Stop Timer to measure execution time
timeTaken=toc;
```

A.2 Rodrigues.m function

```
function    [out,dout]=rodrigues(in)

% RODRIGUES Transform rotation matrix into rotation vector and viceversa.
%
%       Sintax:  [OUT]=RODRIGUES(IN)
%       If IN is a 3x3 rotation matrix then OUT is the
%       corresponding 3x1 rotation vector
%       if IN is a rotation 3-vector then OUT is the
%       corresponding 3x3 rotation matrix
%


%%
%%      Copyright (c) March 1993 -- Pietro Perona
```

```
%%      California Institute of Technology
%%

%% ALL CHECKED BY JEAN-YVES BOUGUET, October 1995.
%% FOR ALL JACOBIAN MATRICES !!! LOOK AT THE TEST AT THE END !!

%% BUG when norm(om)=pi fixed -- April 6th, 1997;
%% Jean-Yves Bouguet

%% Add projection of the 3x3 matrix onto the set of special ortogonal
matrices SO(3) by SVD -- February 7th, 2003;
%% Jean-Yves Bouguet

% BUG FOR THE CASE norm(om)=pi fixed by Mike Burl on Feb 27, 2007


[m,n] = size(in);
%bigeps = 10e+4*eps;
bigeps = 10e+20*eps;

if ((m==1) & (n==3)) | ((m==3) & (n==1)) %% it is a rotation vector
    theta = norm(in);
    if theta < eps
        R = eye(3);

        %if nargout > 1,

        dRdin = [0 0 0;
            0 0 1;
            0 -1 0;
            0 0 -1;
            0 0 0;
            1 0 0;
            0 1 0;
            -1 0 0;
            0 0 0];

        %end;

    else
        if n==length(in)  in=in'; end;  %% make it a column vec. if necess.

        %m3 = [in,theta]

        dm3din = [eye(3);in'/theta];

        omega = in/theta;

        %m2 = [omega;theta]

        dm2dm3 = [eye(3)/theta -in/theta^2; zeros(1,3) 1];

        alpha = cos(theta);
```

```
        beta = sin(theta);
        gamma = 1-cos(theta);
        omegav=[[0   -omega(3)   omega(2)];[omega(3)   0   -omega(1)];[-omega(2)
omega(1)  0 ]];
        A = omega*omega';

        %m1 = [alpha;beta;gamma;omegav;A];

        dm1dm2 = zeros(21,4);
        dm1dm2(1,4) = -sin(theta);
        dm1dm2(2,4) = cos(theta);
        dm1dm2(3,4) = sin(theta);
        dm1dm2(4:12,1:3) = [0 0 0 0 0 1 0 -1 0;
            0 0 -1 0 0 0 1 0 0;
            0 1 0 -1 0 0 0 0 0]';

        w1 = omega(1);
        w2 = omega(2);
        w3 = omega(3);

        dm1dm2(13:21,1) = [2*w1;w2;w3;w2;0;0;w3;0;0];
        dm1dm2(13: 21,2) = [0;w1;0;w1;2*w2;w3;0;w3;0];
        dm1dm2(13:21,3) = [0;0;w1;0;0;w2;w1;w2;2*w3];

        R = eye(3)*alpha + omegav*beta + A*gamma;

        dRdm1 = zeros(9,21);

        dRdm1([1 5 9],1) = ones(3,1);
        dRdm1(:,2) = omegav(:);
        dRdm1(:,4:12) = beta*eye(9);
        dRdm1(:,3) = A(:);
        dRdm1(:,13:21) = gamma*eye(9);

        dRdin = dRdm1 * dm1dm2 * dm2dm3 * dm3din;


    end;
    out = R;
    dout = dRdin;

    %% it is prob. a rot matr.
elseif ((m==n) & (m==3) & (norm(in' * in - eye(3)) < bigeps)...
        & (abs(det(in)-1) < bigeps))
    R = in;

    % project the rotation matrix to SO(3);
    [U,S,V] = svd(R);
    R = U*V';

    tr = (trace(R)-1)/2;
    dtrdR = [1 0 0 0 1 0 0 0 1]/2;
    theta = real(acos(tr));
```

```
if sin(theta) >= 1e-4,

    dthetadtr = -1/sqrt(1-tr^2);

    dthetadR = dthetadtr * dtrdR;
    % var1 = [vth;theta];
    vth = 1/(2*sin(theta));
    dvthdtheta = -vth*cos(theta)/sin(theta);
    dvar1dtheta = [dvthdtheta;1];

    dvar1dR =  dvar1dtheta * dthetadR;


    om1 = [R(3,2)-R(2,3), R(1,3)-R(3,1), R(2,1)-R(1,2)]';

    dom1dR = [0 0 0 0 0 1 0 -1 0;
        0 0 -1 0 0 0 1 0 0;
        0 1 0 -1 0 0 0 0 0];

    % var = [om1;vth;theta];
    dvardR = [dom1dR;dvar1dR];

    % var2 = [om;theta];
    om = vth*om1;
    domdvar = [vth*eye(3) om1 zeros(3,1)];
    dthetadvar = [0 0 0 0 1];
    dvar2dvar = [domdvar;dthetadvar];


    out = om*theta;
    domegadvar2 = [theta*eye(3) om];

    dout = domegadvar2 * dvar2dvar * dvardR;


else
    if tr > 0;          % case norm(om)=0;

        out = [0 0 0]';

        dout = [0 0 0 0 0 1/2 0 -1/2 0;
            0 0 -1/2 0 0 0 1/2 0 0;
            0 1/2 0 -1/2 0 0 0 0 0];
    else

        % case norm(om)=pi;
        if(0)

            %% fixed April 6th by Bouguet -- not working in all cases!
            out = theta * (sqrt((diag(R)+1)/2).*[1;2*(R(1,2:3)>=0)'-1]);
            %keyboard;
```

```
            else

                % Solution by Mike Burl on Feb 27, 2007
                % This is a better way to determine the signs of the
                % entries of the rotation vector using a hash table on all
                % the combinations of signs of a pairs of products (in the
                % rotation matrix)

                % Define hashvec and Smat
                hashvec = [0; -1; -3; -9; 9; 3; 1; 13; 5; -7; -11];
                Smat = [1,1,1; 1,0,-1; 0,1,-1; 1,-1,0; 1,1,0; 0,1,1; 1,0,1;
1,1,1; 1,1,-1;
                    1,-1,-1; 1,-1,1];

                M = (R+eye(3,3))/2;
                uabs = sqrt(M(1,1));
                vabs = sqrt(M(2,2));
                wabs = sqrt(M(3,3));

                mvec = [M(1,2), M(2,3), M(1,3)];
                syn  = ((mvec > 1e-4) - (mvec < -1e-4)); % robust sign()
function
                hash = syn * [9; 3; 1];
                idx = find(hash == hashvec);
                svec = Smat(idx,:)';

                out = theta * [uabs; vabs; wabs] .* svec;

            end;

            if nargout > 1,
                fprintf(1,'WARNING!!!! Jacobian domdR undefined!!!\n');
                dout = NaN*ones(3,9);
            end;
        end;
    end;

else
    error('Neither a rotation matrix nor a rotation vector were provided');
end;

return;

%% test of the Jacobians:

%%%% TEST OF dRdom:
om = randn(3,1);
dom = randn(3,1)/1000000;

[R1,dR1] = rodrigues(om);
R2 = rodrigues(om+dom);

R2a = R1 + reshape(dR1 * dom,3,3);
```

```
gain = norm(R2 - R1)/norm(R2 - R2a)

%%% TEST OF dOmdR:
om = randn(3,1);
R = rodrigues(om);
dom = randn(3,1)/10000;
dR = rodrigues(om+dom) - R;

[omc,domdR] = rodrigues(R);
[om2] = rodrigues(R+dR);

om_app = omc + domdR*dR(:);

gain = norm(om2 - omc)/norm(om2 - om_app)


%%% OTHER BUG: (FIXED NOW!!!)

omu = randn(3,1);
omu = omu/norm(omu)
om = pi*omu;
[R,dR]= rodrigues(om);
[om2] = rodrigues(R);
[om om2]

%%% NORMAL OPERATION

om = randn(3,1);
[R,dR]= rodrigues(om);
[om2] = rodrigues(R);
[om om2]

return

% Test: norm(om) = pi

u = randn(3,1);
u = u / sqrt(sum(u.^2));
om = pi*u;
R = rodrigues(om);

R2 = rodrigues(rodrigues(R));

norm(R - R2)
```

A.3 Skew3.m function

```
function [V,dV] = skew3(v)
%SKEW3      [V,dV] = skew3(v)
%       Takes a 3 components vector and calculates
%       the corresponding skew-symmetric matrix.
%       It is useful for implementing the vector
%       product of 3-vectors: v x u = skew3(v) * u
%
%       dV (optional) returns the 9x3 matrix which represents
%       the 3x3x3 tensor of derivatives of V wrt v.
%

%   Updated 8/30/93

V = zeros(3,3);
V = [[0,-v(3),v(2)]; [v(3),0,-v(1)]; [-v(2),v(1),0]];

if (nargout >=2),
    dV = [0  0  0 ;
    0  0 -1 ;
    0  1  0 ;
    0  0  1 ;
    0  0  0 ;
    -1 0  0 ;
    0 -1  0 ;
    1  0  0 ;
    0  0  0 ];
end;

return;

v = rand(3,1);
eps = 1e-6;
for j=1:3,
    vp = v;
    vp(j) = v(j)+eps;
    dVtest(:,j) = qtoQ(1/eps*(skew3(vp) - skew3(v)));
end;

return;

% difference test
epsilon = 1e-6;
csi = randn(3,1);
rho = randn;
for (k = 1:3),
    csip = csi;
    csip(k) = csip(k)+epsilon;
    diff =  (skew3(csip)-skew3(csi))/epsilon;
    diff = diff';
    dFdcsi_test(:,k) = diff(:);
end;
```

```
[F,dFdcsi] = skew3(csi);
dFdcsi-dFdcsi_test,
norm(ans)
```

A.4 Rigid_motion.m function

```
function [Y,dYdom,dYdT] = rigid_motion(X,om,T);

%rigid_motion.m
%
%[Y,dYdom,dYdT] = rigid_motion(X,om,T)
%
%Computes the rigid motion transformation Y = R*X+T, where R = rodrigues(om).
%
%INPUT: X: 3D structure in the world coordinate frame (3xN matrix for N
points)
%         (om,T): Rigid motion parameters between world coordinate frame and
camera reference frame
%                om: rotation vector (3x1 vector); T: translation vector (3x1
vector)
%
%OUTPUT: Y: 3D coordinates of the structure points in the camera reference
frame (3xN matrix for N points)
%        dYdom: Derivative of Y with respect to om ((3N)x3 matrix)
%        dYdT: Derivative of Y with respect to T ((3N)x3 matrix)
%
%Definitions:
%Let P be a point in 3D of coordinates X in the world reference frame (stored
in the matrix X)
%The coordinate vector of P in the camera reference frame is: Y = R*X + T
%where R is the rotation matrix corresponding to the rotation vector om: R =
rodrigues(om);
%
%Important function called within that program:
%
%rodrigues.m: Computes the rotation matrix corresponding to a rotation vector



if nargin < 3,
   T = zeros(3,1);
   if nargin < 2,
      om = zeros(3,1);
      if nargin < 1,
         error('Need at least a 3D structure as input (in rigid_motion.m)');
         return;
      end;
   end;
end;


[R,dRdom] = rodrigues(om);
```

```
[m,n] = size(X);

Y = R*X + repmat(T,[1 n]);

if nargout > 1,


dYdR = zeros(3*n,9);
dYdT = zeros(3*n,3);

dYdR(1:3:end,1:3:end) =  X';
dYdR(2:3:end,2:3:end) =  X';
dYdR(3:3:end,3:3:end) =  X';

dYdT(1:3:end,1) =  ones(n,1);
dYdT(2:3:end,2) =  ones(n,1);
dYdT(3:3:end,3) =  ones(n,1);

dYdom = dYdR * dRdom;

end;
```

A.5 Project_points2.m function

```
function [xp,dxpdom,dxpdT,dxpdf,dxpdc,dxpdk,dxpdalpha] =
project_points2(X,om,T,f,c,k,alpha)

%project_points2.m
%
%[xp,dxpdom,dxpdT,dxpdf,dxpdc,dxpdk] = project_points2(X,om,T,f,c,k,alpha)
%
%Projects a 3D structure onto the image plane.
%
%INPUT: X: 3D structure in the world coordinate frame (3xN matrix for N
points)
%        (om,T): Rigid motion parameters between world coordinate frame and
camera reference frame
%                om: rotation vector (3x1 vector); T: translation vector (3x1
vector)
%         f: camera focal length in units of horizontal and vertical pixel
units (2x1 vector)
%       c: principal point location in pixel units (2x1 vector)
%       k: Distortion coefficients (radial and tangential) (4x1 vector)
%        alpha: Skew coefficient between x and y pixel (alpha = 0 <=> square
pixels)
%
%OUTPUT: xp: Projected pixel coordinates (2xN matrix for N points)
%        dxpdom: Derivative of xp with respect to om ((2N)x3 matrix)
%        dxpdT: Derivative of xp with respect to T ((2N)x3 matrix)
```

```
%         dxpdf: Derivative of xp with respect to f ((2N)x2 matrix if f is
2x1, or (2N)x1 matrix is f is a scalar)
%         dxpdc: Derivative of xp with respect to c ((2N)x2 matrix)
%         dxpdk: Derivative of xp with respect to k ((2N)x4 matrix)
%
%Definitions:
%Let P be a point in 3D of coordinates X in the world reference frame (stored
in the matrix X)
%The coordinate vector of P in the camera reference frame is: Xc = R*X + T
%where R is the rotation matrix corresponding to the rotation vector om: R =
rodrigues(om);
%call x, y and z the 3 coordinates of Xc: x = Xc(1); y = Xc(2); z = Xc(3);
%The pinehole projection coordinates of P is [a;b] where a=x/z and b=y/z.
%call r^2 = a^2 + b^2.
%The distorted point coordinates are: xd = [xx;yy] where:
%
%xx = a * (1 + kc(1)*r^2 + kc(2)*r^4 + kc(5)*r^6)      +      2*kc(3)*a*b +
kc(4)*(r^2 + 2*a^2);
%yy = b * (1 + kc(1)*r^2 + kc(2)*r^4 + kc(5)*r^6)      +      kc(3)*(r^2 +
2*b^2) + 2*kc(4)*a*b;
%
%The left terms correspond to radial distortion (6th degree), the right terms
correspond to tangential distortion
%
%Finally, convertion into pixel coordinates: The final pixel coordinates
vector xp=[xxp;yyp] where:
%
%xxp = f(1)*(xx + alpha*yy) + c(1)
%yyp = f(2)*yy + c(2)
%
%
%NOTE: About 90 percent of the code takes care fo computing the Jacobian
matrices
%
%
%Important function called within that program:
%
%rodrigues.m: Computes the rotation matrix corresponding to a rotation vector
%
%rigid_motion.m: Computes the rigid motion transformation of a given
structure


if nargin < 7,
    alpha = 0;
    if nargin < 6,
        k = zeros(5,1);
        if nargin < 5,
            c = zeros(2,1);
            if nargin < 4,
                f = ones(2,1);
                if nargin < 3,
                    T = zeros(3,1);
                    if nargin < 2,
                        om = zeros(3,1);
                        if nargin < 1,
```

```
                            error('Need at least a 3D structure to project
(in project_points.m)');
                            return;
                        end;
                    end;
                end;
            end;
        end;
    end;
end;


[m,n] = size(X);

if nargout > 1,
    [Y,dYdom,dYdT] = rigid_motion(X,om,T);
else
    Y = rigid_motion(X,om,T);
end;


inv_Z = 1./Y(3,:);

x = (Y(1:2,:) .* (ones(2,1) * inv_Z)) ;


bb = (-x(1,:) .* inv_Z)'*ones(1,3);
cc = (-x(2,:) .* inv_Z)'*ones(1,3);

if nargout > 1,
    dxdom = zeros(2*n,3);
    dxdom(1:2:end,:) = ((inv_Z')*ones(1,3)) .* dYdom(1:3:end,:) + bb .*
dYdom(3:3:end,:);
    dxdom(2:2:end,:) = ((inv_Z')*ones(1,3)) .* dYdom(2:3:end,:) + cc .*
dYdom(3:3:end,:);

    dxdT = zeros(2*n,3);
    dxdT(1:2:end,:) = ((inv_Z')*ones(1,3)) .* dYdT(1:3:end,:) + bb .*
dYdT(3:3:end,:);
    dxdT(2:2:end,:) = ((inv_Z')*ones(1,3)) .* dYdT(2:3:end,:) + cc .*
dYdT(3:3:end,:);
end;


% Add distortion:

r2 = x(1,:).^2 + x(2,:).^2;

if nargout > 1,
    dr2dom = 2*((x(1,:)')*ones(1,3)) .* dxdom(1:2:end,:) +
2*((x(2,:)')*ones(1,3)) .* dxdom(2:2:end,:);
    dr2dT = 2*((x(1,:)')*ones(1,3)) .* dxdT(1:2:end,:) +
2*((x(2,:)')*ones(1,3)) .* dxdT(2:2:end,:);
end;
```

```
r4 = r2.^2;

if nargout > 1,
    dr4dom = 2*((r2')*ones(1,3)) .* dr2dom;
    dr4dT = 2*((r2')*ones(1,3)) .* dr2dT;
end

r6 = r2.^3;

if nargout > 1,
    dr6dom = 3*((r2'.^2)*ones(1,3)) .* dr2dom;
    dr6dT = 3*((r2'.^2)*ones(1,3)) .* dr2dT;
end;

% Radial distortion:

cdist = 1 + k(1) * r2 + k(2) * r4 + k(5) * r6;

if nargout > 1,
    dcdistdom = k(1) * dr2dom + k(2) * dr4dom + k(5) * dr6dom;
    dcdistdT = k(1) * dr2dT + k(2) * dr4dT + k(5) * dr6dT;
    dcdistdk = [ r2' r4' zeros(n,2) r6'];
end;

xd1 = x .* (ones(2,1)*cdist);

if nargout > 1,
    dxd1dom = zeros(2*n,3);
    dxd1dom(1:2:end,:) = (x(1,:)'*ones(1,3)) .* dcdistdom;
    dxd1dom(2:2:end,:) = (x(2,:)'*ones(1,3)) .* dcdistdom;
    coeff = (reshape([cdist;cdist],2*n,1)*ones(1,3));
    dxd1dom = dxd1dom + coeff.* dxdom;

    dxd1dT = zeros(2*n,3);
    dxd1dT(1:2:end,:) = (x(1,:)'*ones(1,3)) .* dcdistdT;
    dxd1dT(2:2:end,:) = (x(2,:)'*ones(1,3)) .* dcdistdT;
    dxd1dT = dxd1dT + coeff.* dxdT;

    dxd1dk = zeros(2*n,5);
    dxd1dk(1:2:end,:) = (x(1,:)'*ones(1,5)) .* dcdistdk;
    dxd1dk(2:2:end,:) = (x(2,:)'*ones(1,5)) .* dcdistdk;
end;


% tangential distortion:

a1 = 2.*x(1,:).*x(2,:);
a2 = r2 + 2*x(1,:).^2;
a3 = r2 + 2*x(2,:).^2;

delta_x = [k(3)*a1 + k(4)*a2 ;
    k(3) * a3 + k(4)*a1];
```

```
%ddelta_xdx = zeros(2*n,2*n);
aa = (2*k(3)*x(2,:)+6*k(4)*x(1,:))'*ones(1,3);
bb = (2*k(3)*x(1,:)+2*k(4)*x(2,:))'*ones(1,3);
cc = (6*k(3)*x(2,:)+2*k(4)*x(1,:))'*ones(1,3);

if nargout > 1,
    ddelta_xdom = zeros(2*n,3);
    ddelta_xdom(1:2:end,:) = aa .* dxdom(1:2:end,:) + bb .* dxdom(2:2:end,:);
    ddelta_xdom(2:2:end,:) = bb .* dxdom(1:2:end,:) + cc .* dxdom(2:2:end,:);

    ddelta_xdT = zeros(2*n,3);
    ddelta_xdT(1:2:end,:) = aa .* dxdT(1:2:end,:) + bb .* dxdT(2:2:end,:);
    ddelta_xdT(2:2:end,:) = bb .* dxdT(1:2:end,:) + cc .* dxdT(2:2:end,:);

    ddelta_xdk = zeros(2*n,5);
    ddelta_xdk(1:2:end,3) = a1';
    ddelta_xdk(1:2:end,4) = a2';
    ddelta_xdk(2:2:end,3) = a3';
    ddelta_xdk(2:2:end,4) = a1';
end;


xd2 = xd1 + delta_x;

if nargout > 1,
    dxd2dom = dxd1dom + ddelta_xdom ;
    dxd2dT = dxd1dT + ddelta_xdT;
    dxd2dk = dxd1dk + ddelta_xdk ;
end;


% Add Skew:

xd3 = [xd2(1,:) + alpha*xd2(2,:);xd2(2,:)];

% Compute: dxd3dom, dxd3dT, dxd3dk, dxd3dalpha
if nargout > 1,
    dxd3dom = zeros(2*n,3);
    dxd3dom(1:2:2*n,:) = dxd2dom(1:2:2*n,:) + alpha*dxd2dom(2:2:2*n,:);
    dxd3dom(2:2:2*n,:) = dxd2dom(2:2:2*n,:);
    dxd3dT = zeros(2*n,3);
    dxd3dT(1:2:2*n,:) = dxd2dT(1:2:2*n,:) + alpha*dxd2dT(2:2:2*n,:);
    dxd3dT(2:2:2*n,:) = dxd2dT(2:2:2*n,:);
    dxd3dk = zeros(2*n,5);
    dxd3dk(1:2:2*n,:) = dxd2dk(1:2:2*n,:) + alpha*dxd2dk(2:2:2*n,:);
    dxd3dk(2:2:2*n,:) = dxd2dk(2:2:2*n,:);
    dxd3dalpha = zeros(2*n,1);
    dxd3dalpha(1:2:2*n,:) = xd2(2,:)';
end;
```

```
% Pixel coordinates:
if length(f)>1,
    xp = xd3 .* (f * ones(1,n))  +  c*ones(1,n);
    if nargout > 1,
        coeff = reshape(f*ones(1,n),2*n,1);
        dxpdom = (coeff*ones(1,3)) .* dxd3dom;
        dxpdT = (coeff*ones(1,3)) .* dxd3dT;
        dxpdk = (coeff*ones(1,5)) .* dxd3dk;
        dxpdalpha = (coeff) .* dxd3dalpha;
        dxpdf = zeros(2*n,2);
        dxpdf(1:2:end,1) = xd3(1,:)';
        dxpdf(2:2:end,2) = xd3(2,:)';
    end;
else
    xp = f * xd3 + c*ones(1,n);
    if nargout > 1,
        dxpdom = f  * dxd3dom;
        dxpdT = f * dxd3dT;
        dxpdk = f  * dxd3dk;
        dxpdalpha = f .* dxd3dalpha;
        dxpdf = xd3(:);
    end;
end;

if nargout > 1,
    dxpdc = zeros(2*n,2);
    dxpdc(1:2:end,1) = ones(n,1);
    dxpdc(2:2:end,2) = ones(n,1);
end;


return;

% Test of the Jacobians:

n = 10;

X = 10*randn(3,n);
om = randn(3,1);
T = [10*randn(2,1);40];
f = 1000*rand(2,1);
c = 1000*randn(2,1);
k = 0.5*randn(5,1);
alpha = 0.01*randn(1,1);

[x,dxdom,dxdT,dxdf,dxdc,dxdk,dxdalpha] = project_points2(X,om,T,f,c,k,alpha);


% Test on om: OK

dom = 0.000000001 * norm(om)*randn(3,1);
om2 = om + dom;

[x2] = project_points2(X,om2,T,f,c,k,alpha);
```

```
x_pred = x + reshape(dxdom * dom,2,n);


norm(x2-x)/norm(x2 - x_pred)


% Test on T: OK!!

dT = 0.0001 * norm(T)*randn(3,1);
T2 = T + dT;

[x2] = project_points2(X,om,T2,f,c,k,alpha);

x_pred = x + reshape(dxdT * dT,2,n);


norm(x2-x)/norm(x2 - x_pred)



% Test on f: OK!!

df = 0.001 * norm(f)*randn(2,1);
f2 = f + df;

[x2] = project_points2(X,om,T,f2,c,k,alpha);

x_pred = x + reshape(dxdf * df,2,n);


norm(x2-x)/norm(x2 - x_pred)


% Test on c: OK!!

dc = 0.01 * norm(c)*randn(2,1);
c2 = c + dc;

[x2] = project_points2(X,om,T,f,c2,k,alpha);

x_pred = x + reshape(dxdc * dc,2,n);

norm(x2-x)/norm(x2 - x_pred)

% Test on k: OK!!

dk = 0.001 * norm(k)*randn(5,1);
k2 = k + dk;

[x2] = project_points2(X,om,T,f,c,k2,alpha);
```

```
x_pred = x + reshape(dxdk * dk,2,n);

norm(x2-x)/norm(x2 - x_pred)


% Test on alpha: OK!!

dalpha = 0.001 * norm(k)*randn(1,1);
alpha2 = alpha + dalpha;

[x2] = project_points2(X,om,T,f,c,k,alpha2);

x_pred = x + reshape(dxdalpha * dalpha,2,n);

norm(x2-x)/norm(x2 - x_pred)
```

A.6 Normalize_pixel.m function

```
function [xn] = normalize_pixel(x_kk,fc,cc,kc,alpha_c)

%normalize
%
%[xn] = normalize_pixel(x_kk,fc,cc,kc,alpha_c)
%
%Computes the normalized coordinates xn given the pixel coordinates x_kk
%and the intrinsic camera parameters fc, cc and kc.
%
%INPUT: x_kk: Feature locations on the images
%       fc: Camera focal length
%       cc: Principal point coordinates
%       kc: Distortion coefficients
%       alpha_c: Skew coefficient
%
%OUTPUT: xn: Normalized feature locations on the image plane (a 2XN matrix)
%
%Important functions called within that program:
%
%comp_distortion_oulu: undistort pixel coordinates.

if nargin < 5,
   alpha_c = 0;
   if nargin < 4;
      kc = [0;0;0;0;0];
      if nargin < 3;
         cc = [0;0];
         if nargin < 2,
            fc = [1;1];
         end;
      end;
   end;
```

```
end;


% First: Subtract principal point, and divide by the focal length:
x_distort = [(x_kk(1,:) - cc(1))/fc(1);(x_kk(2,:) - cc(2))/fc(2)];

% Second: undo skew
x_distort(1,:) = x_distort(1,:) - alpha_c * x_distort(2,:);

if norm(kc) ~= 0,
    % Third: Compensate for lens distortion:
    xn = comp_distortion_oulu(x_distort,kc);
else
   xn = x_distort;
end;
```

A.7 dAB.m Function

```
function [dABdA,dABdB] = dAB(A,B);

%       [dABdA,dABdB] = dAB(A,B);
%
%       returns : dABdA and dABdB

[p,n] = size(A); [n2,q] = size(B);

if n2 ~= n,
   error(' A and B must have equal inner dimensions');
end;

if issparse(A) |  issparse(B) | p*q*p*n>625 ,
  dABdA=spalloc(p*q,p*n,p*q*n);
else
  dABdA=zeros(p*q,p*n);
end;


for i=1:q,
   for j=1:p,
   ij = j + (i-1)*p;
      for k=1:n,
         kj = j + (k-1)*p;
         dABdA(ij,kj) = B(k,i);
      end;
   end;
end;


if issparse(A) |  issparse(B) | p*q*n*q>625 ,
  dABdB=spalloc(p*q,n*q,p*q*n);
```

```
else
  dABdB=zeros(p*q,q*n);
end;

for i=1:q
   dABdB([i*p-p+1:i*p]',[i*n-n+1:i*n]) = A;
end;
```

A.8 Comp_distortion_oulu.m function

```
function [x] = comp_distortion_oulu(xd,k);

%comp_distortion_oulu.m
%
%[x] = comp_distortion_oulu(xd,k)
%
%Compensates for radial and tangential distortion. Model From Oulu
university.
%For more informatino about the distortion model, check the forward
projection mapping function:
%project_points.m
%
%INPUT: xd: distorted (normalized) point coordinates in the image plane (2xN
matrix)
%       k: Distortion coefficients (radial and tangential) (4x1 vector)
%
%OUTPUT: x: undistorted (normalized) point coordinates in the image plane
(2xN matrix)
%
%Method: Iterative method for compensation.
%
%NOTE: This compensation has to be done after the subtraction
%      of the principal point, and division by the focal length.


if length(k) == 1,

    [x] = comp_distortion(xd,k);

else

    k1 = k(1);
    k2 = k(2);
    k3 = k(5);
    p1 = k(3);
    p2 = k(4);

    x = xd;                 % initial guess

    for kk=1:20,
```

```
        r_2 = sum(x.^2);
        k_radial =  1 + k1 * r_2 + k2 * r_2.^2 + k3 * r_2.^3;
        delta_x = [2*p1*x(1,:).*x(2,:) + p2*(r_2 + 2*x(1,:).^2);
        p1 * (r_2 + 2*x(2,:).^2)+2*p2*x(1,:).*x(2,:)];
        x = (xd - delta_x)./(ones(2,1)*k_radial);

    end;

end;
```

A.9 Correlation based similarity measure-Sum of Absolute Differences (SAD)-Right to Left matching

```
% ************************************************************************
% Title: Function-Compute Correlation between two images using the
% similarity measure of Sum of Absolute Differences (SAD) with Right Image
% as reference.
% Author: Siddhant Ahuja
% Created: May 2008
% Copyright Siddhant Ahuja, 2008
% Inputs: Left Image (var: leftImage), Right Image (var: rightImage),
% Window Size (var: windowSize), Minimum Disparity (dispMin), Maximum
% Disparity (dispMax)
% Outputs: Disparity Map (var: dispMap), Time taken (var: timeTaken)
%         Example         Usage         of         Function:         [dispMap,
timeTaken]=funcSADR2L('StereogramLeft.jpg', 'StereogramRight.jpg', 9, 0, 16);
% ************************************************************************
function [dispMap, timeTaken]=funcSADR2L(leftImage, rightImage, windowSize,
dispMin, dispMax)
try
    % Grab the image information (metadata) of left image using the function
imfinfo
    leftImageInfo=imfinfo(leftImage);
    % Since SADR2L is applied on a grayscale image, determine if the
    % input left image is already in grayscale or color
    if(getfield(leftImageInfo,'ColorType')=='truecolor')
    % Read an image using imread function, convert from RGB color space to
    % grayscale using rgb2gray function and assign it to variable leftImage
        leftImage=rgb2gray(imread(leftImage));
        % Convert the image from uint8 to double
        leftImage=double(leftImage);
    else if(getfield(leftImageInfo,'ColorType')=='grayscale')
    % If the image is already in grayscale, then just read it.
            leftImage=imread(leftImage);
            % Convert the image from uint8 to double
            leftImage=double(leftImage);
        else
            error('The Color Type of Left Image is not acceptable. Acceptable
color types are truecolor or grayscale.');
        end
```

```
    end
catch
    % if it is not an image but a variable
    leftImage=leftImage;
end
try
    % Grab the image information (metadata) of right image using the function
imfinfo
    rightImageInfo=imfinfo(rightImage);
    % Since SADR2L is applied on a grayscale image, determine if the
    % input right image is already in grayscale or color
    if(getfield(rightImageInfo,'ColorType')=='truecolor')
    % Read an image using imread function, convert from RGB color space to
    % grayscale using rgb2gray function and assign it to variable rightImage
        rightImage=rgb2gray(imread(rightImage));
        % Convert the image from uint8 to double
        rightImage=double(rightImage);
    else if(getfield(rightImageInfo,'ColorType')=='grayscale')
    % If the image is already in grayscale, then just read it.
            rightImage=imread(rightImage);
            % Convert the image from uint8 to double
            rightImage=double(rightImage);
        else
            error('The Color Type of Right Image is not acceptable.
Acceptable color types are truecolor or grayscale.');
        end
    end
catch
    % if it is not an image but a variable
    rightImage=rightImage;
end
% Find the size (columns and rows) of the left image and assign the rows to
% variable nrLeft, and columns to variable ncLeft
[nrLeft,ncLeft] = size(leftImage);
% Find the size (columns and rows) of the right image and assign the rows to
% variable nrRight, and columns to variable ncRight
[nrRight,ncRight] = size(rightImage);
% Check to see if both the left and right images have same number of rows
% and columns
if(nrLeft==nrRight && ncLeft==ncRight)
else
    error('Both left and right images should have the same number of rows and
columns');
end
% Check the size of window to see if it is an odd number.
if (mod(windowSize,2)==0)
    error('The window size must be an odd number.');
end
% Check whether minimum disparity is less than the maximum disparity.
if (dispMin>dispMax)
    error('Minimum Disparity must be less than the Maximum disparity.');
end
% Create an image of size nrLeft and ncLeft, fill it with zeros and assign
% it to variable dispMap
dispMap=zeros(nrLeft, ncLeft);
% Find out how many rows and columns are to the left/right/up/down of the
% central pixel based on the window size
```

```
win=(windowSize-1)/2;
tic; % Initialize the timer to calculate the time consumed.
for(i=1+win:1:nrLeft-win)
    for(j=1+win:1:ncLeft-win-dispMax)
        prevSAD = 65532;
        temp=0.0;
        bestMatchSoFar = dispMin;
        for(dispRange=dispMin:1:dispMax)
            sad=0.0;
            for(a=-win:1:win)
                for(b=-win:1:win)
                    if (j+b+dispRange <= ncLeft)
                        temp=rightImage(i+a,j+b)-
leftImage(i+a,j+b+dispRange);
                        if(temp<0.0)
                            temp=temp*-1.0;
                        end
                        sad=sad+temp;
                    end
                end
            end
            if (prevSAD > sad)
                prevSAD = sad;
                bestMatchSoFar = dispRange;
            end
        end
        dispMap(i,j) = bestMatchSoFar;
    end
end
% Stop the timer to calculate the time consumed.
timeTaken=toc;
```

A.10 Correlation based similarity measure-Sum of Absolute Differences (SAD)-Left to Right matching

```
% ************************************************************************
% Title: Function-Compute Correlation between two images using the
% similarity measure of Sum of Absolute Differences (SAD) with Left Image
% as reference.
% Author: Siddhant Ahuja
% Created: May 2008
% Copyright Siddhant Ahuja, 2008
% Inputs: Left Image (var: leftImage), Right Image (var: rightImage),
% Window Size (var: windowSize), Minimum Disparity (dispMin), Maximum
% Disparity (dispMax)
% Outputs: Disparity Map (var: dispMap), Time taken (var: timeTaken)
%         Example         Usage         of         Function:         [dispMap,
timeTaken]=funcSADL2R('StereogramLeft.jpg', 'StereogramRight.jpg', 9, 0, 16);
% ************************************************************************
function [dispMap, timeTaken]=funcSADL2R(leftImage, rightImage, windowSize,
dispMin, dispMax)
```

```
try
    % Grab the image information (metadata) of left image using the function
imfinfo
    leftImageInfo=imfinfo(leftImage);
    % Since SADL2R is applied on a grayscale image, determine if the
    % input left image is already in grayscale or color
    if(getfield(leftImageInfo,'ColorType')=='truecolor')
    % Read an image using imread function, convert from RGB color space to
    % grayscale using rgb2gray function and assign it to variable leftImage
        leftImage=rgb2gray(imread(leftImage));
        % Convert the image from uint8 to double
        leftImage=double(leftImage);
    else if(getfield(leftImageInfo,'ColorType')=='grayscale')
    % If the image is already in grayscale, then just read it.
            leftImage=imread(leftImage);
            % Convert the image from uint8 to double
            leftImage=double(leftImage);
        else
            error('The Color Type of Left Image is not acceptable. Acceptable
color types are truecolor or grayscale.');
        end
    end
catch
    % if it is not an image but a variable
    leftImage=leftImage;
end
try
    % Grab the image information (metadata) of right image using the function
imfinfo
    rightImageInfo=imfinfo(rightImage);
    % Since SADL2R is applied on a grayscale image, determine if the
    % input right image is already in grayscale or color
    if(getfield(rightImageInfo,'ColorType')=='truecolor')
    % Read an image using imread function, convert from RGB color space to
    % grayscale using rgb2gray function and assign it to variable rightImage
        rightImage=rgb2gray(imread(rightImage));
        % Convert the image from uint8 to double
        rightImage=double(rightImage);
    else if(getfield(rightImageInfo,'ColorType')=='grayscale')
    % If the image is already in grayscale, then just read it.
            rightImage=imread(rightImage);
            % Convert the image from uint8 to double
            rightImage=double(rightImage);
        else
            error('The  Color  Type  of  Right  Image  is  not  acceptable.
Acceptable color types are truecolor or grayscale.');
        end
    end
catch
    % if it is not an image but a variable
    rightImage=rightImage;
end
% Find the size (columns and rows) of the left image and assign the rows to
% variable nrLeft, and columns to variable ncLeft
[nrLeft,ncLeft] = size(leftImage);
% Find the size (columns and rows) of the right image and assign the rows to
% variable nrRight, and columns to variable ncRight
```

```
[nrRight,ncRight] = size(rightImage);
% Check to see if both the left and right images have same number of rows
% and columns
if(nrLeft==nrRight && ncLeft==ncRight)
else
    error('Both left and right images should have the same number of rows and
columns');
end
% Check the size of window to see if it is an odd number.
if (mod(windowSize,2)==0)
    error('The window size must be an odd number.');
end
% Check whether minimum disparity is less than the maximum disparity.
if (dispMin>dispMax)
    error('Minimum Disparity must be less than the Maximum disparity.');
end
% Create an image of size nrLeft and ncLeft, fill it with zeros and assign
% it to variable dispMap
dispMap=zeros(nrLeft, ncLeft);
% Find out how many rows and columns are to the left/right/up/down of the
% central pixel based on the window size
win=(windowSize-1)/2;
tic; % Initialize the timer to calculate the time consumed.
for(i=1+win:1:nrLeft-win)
    for(j=1+win+dispMax:1:ncLeft-win)
        prevSAD = 65532;
        temp=0.0;
        bestMatchSoFar = dispMin;
        for(dispRange=-dispMin:-1:-dispMax)
            sad=0.0;
            for(a=-win:1:win)
                for(b=-win:1:win)
                    if (j-win+dispRange > 0)
                        temp=leftImage(i+a,j+b)-
rightImage(i+a,j+b+dispRange);
                        if(temp<0.0)
                            temp=temp*-1.0;
                        end
                        sad=sad+temp;
                    end
                end
            end
            if (prevSAD > sad)
                prevSAD = sad;
                bestMatchSoFar = dispRange;
            end
        end
        dispMap(i,j) = -bestMatchSoFar;
    end
end
% Stop the timer to calculate the time consumed.
timeTaken=toc;
```

A.11 Correlation based similarity measure-Sum of Squared Differences (SSD)-Right to Left matching

```
% ************************************************************************
% Title: Function-Compute Correlation between two images using the
% similarity measure of Sum of Squared Differences (SSD) with Right Image
% as reference.
% Author: Siddhant Ahuja
% Created: May 2008
% Copyright Siddhant Ahuja, 2008
% Inputs: Left Image (var: leftImage), Right Image (var: rightImage),
% Window Size (var: windowSize), Minimum Disparity (dispMin), Maximum
% Disparity (dispMax)
% Outputs: Disparity Map (var: dispMap), Time taken (var: timeTaken)
%         Example         Usage         of         Function:         [dispMap,
timeTaken]=funcSSDR2L('StereogramLeft.jpg', 'StereogramRight.jpg', 9, 0, 16);
% ************************************************************************
function [dispMap, timeTaken]=funcSSDR2L(leftImage, rightImage, windowSize,
dispMin, dispMax)
try
    % Grab the image information (metadata) of left image using the function
imfinfo
    leftImageInfo=imfinfo(leftImage);
    % Since SSDR2L is applied on a grayscale image, determine if the
    % input left image is already in grayscale or color
    if(getfield(leftImageInfo,'ColorType')=='truecolor')
    % Read an image using imread function, convert from RGB color space to
    % grayscale using rgb2gray function and assign it to variable leftImage
        leftImage=rgb2gray(imread(leftImage));
        % Convert the image from uint8 to double
        leftImage=double(leftImage);
    else if(getfield(leftImageInfo,'ColorType')=='grayscale')
    % If the image is already in grayscale, then just read it.
            leftImage=imread(leftImage);
            % Convert the image from uint8 to double
            leftImage=double(leftImage);
        else
            error('The Color Type of Left Image is not acceptable. Acceptable
color types are truecolor or grayscale.');
        end
    end
catch
    % if it is not an image but a variable
    leftImage=leftImage;
end
try
    % Grab the image information (metadata) of right image using the function
imfinfo
    rightImageInfo=imfinfo(rightImage);
    % Since SSDR2L is applied on a grayscale image, determine if the
    % input right image is already in grayscale or color
    if(getfield(rightImageInfo,'ColorType')=='truecolor')
    % Read an image using imread function, convert from RGB color space to
```

```
    % grayscale using rgb2gray function and assign it to variable rightImage
        rightImage=rgb2gray(imread(rightImage));
        % Convert the image from uint8 to double
        rightImage=double(rightImage);
    else if(getfield(rightImageInfo,'ColorType')=='grayscale')
    % If the image is already in grayscale, then just read it.
            rightImage=imread(rightImage);
            % Convert the image from uint8 to double
            rightImage=double(rightImage);
        else
            error('The Color Type of Right Image is not acceptable.
Acceptable color types are truecolor or grayscale.');
        end
    end
catch
    % if it is not an image but a variable
    rightImage=rightImage;
end
% Find the size (columns and rows) of the left image and assign the rows to
% variable nrLeft, and columns to variable ncLeft
[nrLeft,ncLeft] = size(leftImage);
% Find the size (columns and rows) of the right image and assign the rows to
% variable nrRight, and columns to variable ncRight
[nrRight,ncRight] = size(rightImage);
% Check to see if both the left and right images have same number of rows
% and columns
if(nrLeft==nrRight && ncLeft==ncRight)
else
    error('Both left and right images should have the same number of rows and
columns');
end
% Check the size of window to see if it is an odd number.
if (mod(windowSize,2)==0)
    error('The window size must be an odd number.');
end
% Check whether minimum disparity is less than the maximum disparity.
if (dispMin>dispMax)
    error('Minimum Disparity must be less than the Maximum disparity.');
end
% Create an image of size nrLeft and ncLeft, fill it with zeros and assign
% it to variable dispMap
dispMap=zeros(nrLeft, ncLeft);
% Find out how many rows and columns are to the left/right/up/down of the
% central pixel based on the window size
win=(windowSize-1)/2;
tic; % Initialize the timer to calculate the time consumed.
for(i=1+win:1:nrLeft-win)
    for(j=1+win:1:ncLeft-win-dispMax)
        prevSSD = 65532;
        temp=0.0;
        bestMatchSoFar = dispMin;
        for(dispRange=dispMin:1:dispMax)
            ssd=0.0;
            for(a=-win:1:win)
                for(b=-win:1:win)
                    if (j+b+dispRange <= ncLeft)
```

```
                        temp=rightImage(i+a,j+b)-
leftImage(i+a,j+b+dispRange);
                        temp=temp*temp;
                        ssd=ssd+temp;
                    end
                end
            end
            if (prevSSD > ssd)
                prevSSD = ssd;
                bestMatchSoFar = dispRange;
            end
        end
        dispMap(i,j) = bestMatchSoFar;
    end
end
% Stop the timer to calculate the time consumed.
timeTaken=toc;
```

A.12 Correlation based similarity measure-Normalized Cross Correlation (NCC)-Right to Left matching

```
% ************************************************************************
% Title: Function-Compute Correlation between two images using the
% similarity measure of Normalized Cross Correlation (NCC) with Right Image
% as reference.
% Author: Siddhant Ahuja
% Created: May 2008
% Copyright Siddhant Ahuja, 2008
% Inputs: Left Image (var: leftImage), Right Image (var: rightImage),
% Window Size (var: windowSize), Minimum Disparity (dispMin), Maximum
% Disparity (dispMax)
% Outputs: Disparity Map (var: dispMap), Time taken (var: timeTaken)
%         Example         Usage         of         Function:         [dispMap,
timeTaken]=funcNCCR2L('StereogramLeft.jpg', 'StereogramRight.jpg', 9, 0, 16);
% ************************************************************************
function [dispMap, timeTaken]=funcNCCR2L(leftImage, rightImage, windowSize,
dispMin, dispMax)
try
    % Grab the image information (metadata) of left image using the function
imfinfo
    leftImageInfo=imfinfo(leftImage);
    % Since NCCR2L is applied on a grayscale image, determine if the
    % input left image is already in grayscale or color
    if(getfield(leftImageInfo,'ColorType')=='truecolor')
    % Read an image using imread function, convert from RGB color space to
    % grayscale using rgb2gray function and assign it to variable leftImage
        leftImage=rgb2gray(imread(leftImage));
        % Convert the image from uint8 to double
        leftImage=double(leftImage);
    else if(getfield(leftImageInfo,'ColorType')=='grayscale')
    % If the image is already in grayscale, then just read it.
```

```
            leftImage=imread(leftImage);
            % Convert the image from uint8 to double
            leftImage=double(leftImage);
        else
            error('The Color Type of Left Image is not acceptable. Acceptable
color types are truecolor or grayscale.');
        end
    end
catch
    % if it is not an image but a variable
    leftImage=leftImage;
end
try
    % Grab the image information (metadata) of right image using the function
imfinfo
    rightImageInfo=imfinfo(rightImage);
    % Since NCCR2L is applied on a grayscale image, determine if the
    % input right image is already in grayscale or color
    if(getfield(rightImageInfo,'ColorType')=='truecolor')
    % Read an image using imread function, convert from RGB color space to
    % grayscale using rgb2gray function and assign it to variable rightImage
        rightImage=rgb2gray(imread(rightImage));
        % Convert the image from uint8 to double
        rightImage=double(rightImage);
    else if(getfield(rightImageInfo,'ColorType')=='grayscale')
    % If the image is already in grayscale, then just read it.
            rightImage=imread(rightImage);
            % Convert the image from uint8 to double
            rightImage=double(rightImage);
        else
            error('The Color Type of Right Image is not acceptable.
Acceptable color types are truecolor or grayscale.');
        end
    end
catch
    % if it is not an image but a variable
    rightImage=rightImage;
end
% Find the size (columns and rows) of the left image and assign the rows to
% variable nrLeft, and columns to variable ncLeft
[nrLeft,ncLeft] = size(leftImage);
% Find the size (columns and rows) of the right image and assign the rows to
% variable nrRight, and columns to variable ncRight
[nrRight,ncRight] = size(rightImage);
% Check to see if both the left and right images have same number of rows
% and columns
if(nrLeft==nrRight && ncLeft==ncRight)
else
    error('Both left and right images should have the same number of rows and
columns');
end
% Check the size of window to see if it is an odd number.
if (mod(windowSize,2)==0)
    error('The window size must be an odd number.');
end
% Check whether minimum disparity is less than the maximum disparity.
if (dispMin>dispMax)
```

```
    error('Minimum Disparity must be less than the Maximum disparity.');
end
% Create an image of size nrLeft and ncLeft, fill it with zeros and assign
% it to variable dispMap
dispMap=zeros(nrLeft, ncLeft);
% Find out how many rows and columns are to the left/right/up/down of the
% central pixel based on the window size
win=(windowSize-1)/2;
tic; % Initialize the timer to calculate the time consumed.
for(i=1+win:1:nrLeft-win)
    for(j=1+win:1:ncLeft-win-dispMax)
        prevNCC = 0.0;
        bestMatchSoFar = dispMin;
        for(dispRange=dispMin:1:dispMax)
            ncc=0.0;
            nccNumerator=0.0;
            nccDenominator=0.0;
            nccDenominatorRightWindow=0.0;
            nccDenominatorLeftWindow=0.0;
            for(a=-win:1:win)
                for(b=-win:1:win)

nccNumerator=nccNumerator+(rightImage(i+a,j+b)*leftImage(i+a,j+b+dispRange));

nccDenominatorRightWindow=nccDenominatorRightWindow+(rightImage(i+a,j+b)*righ
tImage(i+a,j+b));

nccDenominatorLeftWindow=nccDenominatorLeftWindow+(leftImage(i+a,j+b+dispRang
e)*leftImage(i+a,j+b+dispRange));
                end
            end

nccDenominator=sqrt(nccDenominatorRightWindow*nccDenominatorLeftWindow);
            ncc=nccNumerator/nccDenominator;
            if (prevNCC < ncc)
                prevNCC = ncc;
                bestMatchSoFar = dispRange;
            end
        end
        dispMap(i,j) = bestMatchSoFar;
    end
end
% Stop the timer to calculate the time consumed.
timeTaken=toc;
```

A.13 Left/Right Consistency (LRC) Check

```
% ************************************************************************
% Title: Function-Left/Right Consistency Check
% Author: Siddhant Ahuja
% Created: May 2008
% Copyright Siddhant Ahuja, 2008
% Inputs: Left Image (var: leftImage), Right Image (var: rightImage),
% Window Size (var: windowSize), Minimum Disparity (dispMin), Maximum
% Disparity (dispMax), Threshold for the check (var: thresh) typically 1.0
% Outputs: Disparity Map (var: dispMap), Time taken (var: timeTaken)
%        Example        Usage        of        Function:        [dispMapLRC,
timeTaken]=funcLRCCheck('TsukubaLeft.jpg', 'TsukubaRight.jpg', 9, 0, 16,2);
% ************************************************************************
function    [dispMapLRC,    timeTaken]=funcLRCCheck(leftImage,    rightImage,
windowSize, dispMin, dispMax, thresh)
% Initiate the Timer to calculate the time consumed.
tic;
% Perform SAD Correlation based matching (Right to Left)
[dispMapR2L,   timeTakenR2L]=funcSADR2L(leftImage,   rightImage,   windowSize,
dispMin, dispMax);
% Perform SAD Correlation based matching (Left to Right)
[dispMapL2R,          timeTakenL2R]=funcSADL2R(leftImage,          rightImage,
windowSize,dispMin , dispMax);
% Prepare matrix for subtraction and scale it for comparison
dispMapL2R=-dispMapL2R;
% Find the size (columns and rows) of the L2R Disparity map and assign the
rows to
% variable nrLRCCheck, and columns to variable ncLRCCheck
[nrLRCCheck,ncLRCCheck] = size(dispMapL2R);
% Create an image of size nrLRCCheck and ncLRCCheck, fill it with zeros and
assign
% it to variable dispMapLRC
dispMapLRC=zeros(nrLRCCheck,ncLRCCheck);
% Find out how many rows and columns are to the left/right/up/down of the
% central pixel based on the window size
win=(windowSize-1)/2;
for(i=1:1:nrLRCCheck)
    for(j=1:1:ncLRCCheck)
        xl=j;
        xr=xl+dispMapL2R(i,xl);
        if (xr>ncLRCCheck||xr<1)
            dispMapLRC(i,j) = 0; %% occluded pixel
        else
            xlp=xr+dispMapR2L(i,xr);
            if (abs(xl-xlp)<thresh)
                dispMapLRC(i,j) = -dispMapL2R(i,j);  %% non-occluded pixel
            else
                dispMapLRC(i,j) = 0; %% occluded pixel
            end
        end
    end
end
% Terminate the Timer to calculate the time consumed.
```

```
timeTaken=toc;
```

A.14 Quality Metric-Root Mean Squared Error (RMS)

```
% ************************************************************************
% Title: Function-Compute Root Mean Squared (RMS) Error
% Author: Siddhant Ahuja
% Created: May 2008
% Copyright Siddhant Ahuja, 2008
% Inputs: Computed Disparity Map (var: computedDisparityMap), Ground Truth
Disparity Map (var: groundTruthDisparityMap),
% How many pixels to ignore at the border (var: borderPixelsToIgnore). For
% a 9x9 windowSize used, border pixels to ignore should be (9-1)/2 or 4
% pixels, Disparity range (var: dispRange), Scale factor for groundtruth
% (var: scale)
% Outputs: RMS Error (var: rmsError), Time taken (var: timeTaken)
%     Example     Usage     of     Function:     [rmsError,     timeTaken]=
RMSError('TsukubaSAD9x9DispRange=0-16.png','TsukubaGroundTruth.png',4,16,8);
% ************************************************************************
function                              [rmsError,                          timeTaken]=
funcRMSError(computedDisparityMap,groundTruthDisparityMap,borderPixelsToIgnor
e,dispRange, scale)
% Read an image using imread function, and assign it to variable
% computedDisparityMap
try
    computedDisparityMap=imread(computedDisparityMap);
catch
    % if it is not an image but a variable
    computedDisparityMap=computedDisparityMap;
end
% Convert the image from uint8 to double
computedDisparityMap=double(computedDisparityMap);
% Read an image using imread function, and assign it to variable
% groundTruthDisparityMap
groundTruthDisparityMap=imread(groundTruthDisparityMap);
% Convert the image from uint8 to double
groundTruthDisparityMap=double(groundTruthDisparityMap);
% Find the size (columns and rows) of the left image and assign the rows to
%    variable    nrComputedDisparityMap,    and    columns    to    variable
ncComputedDisparityMap
[nrComputedDisparityMap,ncComputedDisparityMap] = size(computedDisparityMap);
% Find the size (columns and rows) of the image and assign the rows to
%    variable    nrGroundTruthDisparityMap,    and    columns    to    variable
ncGroundTruthDisparityMap
[nrGroundTruthDisparityMap,ncGroundTruthDisparityMap]                          =
size(groundTruthDisparityMap);
% Check to see if both the left and right images have same number of rows
% and columns
if(nrComputedDisparityMap==nrGroundTruthDisparityMap                          &&
ncComputedDisparityMap==ncGroundTruthDisparityMap)
else
```

```
    error('Both Computed Disparity Map and Groundtruth Disparity Map images
should have the same number of rows and columns');
end
numPixels=0;
rmsError=0.0;
tic; % Initialize the timer to calculate the time consumed.
% Calculate rms error
for (i=borderPixelsToIgnore:1:nrComputedDisparityMap-borderPixelsToIgnore)
    for(j=borderPixelsToIgnore:1:ncComputedDisparityMap-borderPixelsToIgnore-
dispRange)
        if(groundTruthDisparityMap(i,j)~=0.0)  % Ignore Pixels with unknown
disparity in the groundTruthDisparityMap
            rmsError=        rmsError+(abs((computedDisparityMap(i,j)*scale)-
groundTruthDisparityMap(i,j))^2);
            numPixels=numPixels+1;
        end
    end
end
rmsError=sqrt(rmsError/numPixels);
% Stop the timer to calculate the time consumed.
timeTaken=toc;
```

A.15 Quality Metric-Percentage of Bad Matching Pixels

```
% ************************************************************************
% Title: Function-Compute Percentage of bad matching pixels
% Author: Siddhant Ahuja
% Created: May 2008
% Copyright Siddhant Ahuja, 2008
% Inputs: Computed Disparity Map (var: computedDisparityMap), Ground Truth
Disparity Map (var: groundTruthDisparityMap),
% How many pixels to ignore at the border (var: borderPixelsToIgnore). For
% a 9x9 windowSize used, border pixels to ignore should be (9-1)/2 or 4
% pixels, Disparity range (var: dispRange), Threshold (var: thresh), Scale
% factor for groundtruth (var: scale)
% Outputs: Percentage Bad matching pixels (var: perBADMatch), Time taken
% (var: timeTaken)
% Example Usage of Function: [perBADMatch, timeTaken]=
% funcPercentBadMatchingPixels(dispMap,'VenusGroundTruthL2R.png',4,16,1,8)
% ************************************************************************
function                          [perBADMatch,                          timeTaken]=
funcPercentBadMatchingPixels(computedDisparityMap,groundTruthDisparityMap,bor
derPixelsToIgnore,dispRange,thresh, scale)
% Read an image using imread function, and assign it to variable
% computedDisparityMap
try
    computedDisparityMap=imread(computedDisparityMap);
catch
    % if it is not an image but a variable
    computedDisparityMap=computedDisparityMap;
end
% Convert the image from uint8 to double
```

```
computedDisparityMap=double(computedDisparityMap);
% Read an image using imread function, and assign it to variable
% groundTruthDisparityMap
groundTruthDisparityMap=imread(groundTruthDisparityMap);
% Convert the image from uint8 to double
groundTruthDisparityMap=double(groundTruthDisparityMap);
% Find the size (columns and rows) of the left image and assign the rows to
%    variable    nrComputedDisparityMap,    and    columns    to    variable
ncComputedDisparityMap
[nrComputedDisparityMap,ncComputedDisparityMap] = size(computedDisparityMap);
% Find the size (columns and rows) of the image and assign the rows to
%    variable    nrGroundTruthDisparityMap,    and    columns    to    variable
ncGroundTruthDisparityMap
[nrGroundTruthDisparityMap,ncGroundTruthDisparityMap]                          =
size(groundTruthDisparityMap);
% Check to see if both the left and right images have same number of rows
% and columns
if(nrComputedDisparityMap==nrGroundTruthDisparityMap                           &&
ncComputedDisparityMap==ncGroundTruthDisparityMap)
else
    error('Both Computed Disparity Map and Groundtruth Disparity Map images
should have the same number of rows and columns');
end
numPixels=0;
perBADMatch=0.0;
tic; % Initialize the timer to calculate the time consumed.
% Calculate Percentage Bad Matching Pixels
for (i=borderPixelsToIgnore:1:nrComputedDisparityMap-borderPixelsToIgnore)
    for(j=borderPixelsToIgnore:1:ncComputedDisparityMap-borderPixelsToIgnore-
dispRange)
        if(groundTruthDisparityMap(i,j)~=0.0)  % Ignore Pixels with unknown
disparity in the groundTruthDisparityMap
            if(abs((computedDisparityMap(i,j)*scale)-
groundTruthDisparityMap(i,j))>thresh)
                perBADMatch=perBADMatch+1;
            end
            numPixels=numPixels+1;
        end
    end
end
perBADMatch=perBADMatch/numPixels;
% Stop the timer to calculate the time consumed.
timeTaken=toc;
```

A.16 Rank Transform

```
% ************************************************************************
% Title: Function-Rank Transform of a given Image
% Author: Siddhant Ahuja
% Created: May 2008
% Copyright Siddhant Ahuja, 2008
% Inputs: Image (var: inputImage), Window size assuming square window (var:
% windowSize)
% Outputs: Rank Tranformed Image (var: rankTransformedImage),
% Time taken (var: timeTaken)
% Example Usage of Function: [a,b]=funcRankOneImage('Img.png', 3)
% ************************************************************************
function [rankTransformedImage, timeTaken] = funcRankOneImage(inputImage,
windowSize)
% Grab the image information (metadata) using the function imfinfo
imageInfo=imfinfo(inputImage);
% Since Rank Transform is applied on a grayscale image, determine if the
% input image is already in grayscale or color
if(getfield(imageInfo,'ColorType')=='truecolor')
% Read an image using imread function, convert from RGB color space to
% grayscale using rgb2gray function and assign it to variable inputImage
    inputImage=rgb2gray(imread(inputImage));
else if(getfield(imageInfo,'ColorType')=='grayscale')
% If the image is already in grayscale, then just read it.
        inputImage=imread(inputImage);
    else
        error('The Color Type of Input Image is not acceptable. Acceptable
color types are truecolor or grayscale.');
    end
end
% Check the size of window to see if it is an odd number.
if (mod(windowSize,2)==0)
    error('The window size must be an odd number.');
end
% Initialize the timer to calculate the time consumed.
tic;
% Find the size (columns and rows) of the image and assign the rows to
% variable nr, and columns to variable nc
[nr,nc] = size(inputImage);
% Create an image of size nr and nc, fill it with zeros and assign
% it to variable rankTransformedImage
rankTransformedImage = zeros(nr,nc);
% Find out how many rows and columns are to the left/right/up/down of the
% central pixel based on the window size
R= (windowSize-1)/2;
for (i=R+1:1:nr-R) % Go through all the rows in an image (minus R at the
borders)
    for (j=R+1:1:nc-R) % Go through all the columns in an image (minus R at
the borders)
        rank = 0; % Initialize default rank to 0
        for (a=-R:1:R) % Within the square window, go through all the rows
            for (b=-R:1:R) % Within the square window, go through all the
columns
```

```
                % If the intensity of the neighboring pixel is less than
                % that of the central pixel, then increase the rank
                if (inputImage(i+a,j+b) < inputImage(i,j))
                    rank=rank+1;
                end
            end
        end
        % Assign the rank value to the pixel in imgTemp
        rankTransformedImage(i,j) = rank;
    end
end
% Stop the timer to calculate the time consumed.
timeTaken=toc;
```

A.17 Census Transform

```
% ************************************************************************
% Title: Function-Census Transform of a given Image
% Author: Siddhant Ahuja
% Created: May 2008
% Copyright Siddhant Ahuja, 2008
% Inputs: Image (var: inputImage), Window size assuming square window (var:
% windowSize) of 3x3 or 5x5 only.
% Outputs: Census Tranformed Image (var: censusTransformedImage),
% Time taken (var: timeTaken)
% Example Usage of Function: [a,b]=funcCensusOneImage('Img.png', 3)
% ************************************************************************
function [censusTransformedImage, timeTaken] = funcCensusOneImage(inputImage,
windowSize)
% Grab the image information (metadata) using the function imfinfo
imageInfo=imfinfo(inputImage);
% Since Census Transform is applied on a grayscale image, determine if the
% input image is already in grayscale or color
if(getfield(imageInfo,'ColorType')=='truecolor')
% Read an image using imread function, convert from RGB color space to
% grayscale using rgb2gray function and assign it to variable inputImage
    inputImage=rgb2gray(imread(inputImage));
else if(getfield(imageInfo,'ColorType')=='grayscale')
% If the image is already in grayscale, then just read it.
        inputImage=imread(inputImage);
    else
        error('The Color Type of Input Image is not acceptable. Acceptable
color types are truecolor or grayscale.');
    end
end
% Find the size (columns and rows) of the image and assign the rows to
% variable nr, and columns to variable nc
[nr,nc] = size(inputImage);
% Check the size of window to see if it is an odd number.
if (mod(windowSize,2)==0)
    error('The window size must be an odd number.');
end
```

```
if (windowSize==3)
    bits=uint8(0);
% Create an image of size nr and nc, fill it with zeros and assign
% it to variable censusTransformedImage of type uint8
    censusTransformedImage=uint8(zeros(nr,nc));
else if (windowSize==5)
        bits=uint32(0);
% Create an image of size nr and nc, fill it with zeros and assign
% it to variable censusTransformedImage of type uint32
        censusTransformedImage=uint32(zeros(nr,nc));
    else
        error('The size of the window is not acceptable. Just 3x3 and 5x5
windows are acceptable.');
    end
end
% Initialize the timer to calculate the time consumed.
tic;
% Find out how many rows and columns are to the left/right/up/down of the
% central pixel
C= (windowSize-1)/2;
for(j=C+1:1:nc-C) % Go through all the columns in an image (minus C at the
borders)
    for(i=C+1:1:nr-C) % Go through all the rows in an image (minus C at the
borders)
       census = 0; % Initialize default census to 0
        for (a=-C:1:C) % Within the square window, go through all the rows
            for (b=-C:1:C) % Within the square window, go through all the
columns
                if (~(a==C+1 && b==C+1)) % Exclude the centre pixel from the
calculation
                    census=bitshift(census,1); %Shift the bits to the left
by 1
                    % If the intensity of the neighboring pixel is less than
                    % that of the central pixel, then add one to the bit
                    % string
                    if (inputImage(i+a,j+b) < inputImage(i,j))
                        census=census+1;
                    end
                end
            end
        end
        % Assign the census bit string value to the pixel in imgTemp
        censusTransformedImage(i,j) = census;
    end
end
% Stop the timer to calculate the time consumed.
timeTaken=toc;
```

A.18 Sum of Hamming Distances-Right to Left Matching

```
% ************************************************************************
% Title: Function-Compute Correlation between two images using the
% similarity measure of Sum of Hamming Distances (SHD) with Right Image
% as reference.
% Author: Siddhant Ahuja
% Created: May 2008
% Copyright Siddhant Ahuja, 2008
% Inputs: Left Image (var: leftImage), Right Image (var: rightImage),
% Window Size (var: windowSize), Minimum Disparity (dispMin), Maximum
% Disparity (dispMax)
% Outputs: Disparity Map (var: dispMap), Time taken (var: timeTaken)
%         Example         Usage         of         Function:         [dispMap,
timeTaken]=funcSHDR2L('TsukubaLeft.jpg', 'TsukubaRight.jpg', 9, 0, 16);
% ************************************************************************
function [dispMap, timeTaken]=funcSHDR2L(leftImage, rightImage, windowSize,
dispMin, dispMax)
try
    % Grab the image information (metadata) of left image using the function
imfinfo
    leftImageInfo=imfinfo(leftImage);
    % Since SHDR2L is applied on a grayscale image, determine if the
    % input left image is already in grayscale or color
    if(getfield(leftImageInfo,'ColorType')=='truecolor')
    % Read an image using imread function, convert from RGB color space to
    % grayscale using rgb2gray function and assign it to variable leftImage
        leftImage=rgb2gray(imread(leftImage));
    else if(getfield(leftImageInfo,'ColorType')=='grayscale')
    % If the image is already in grayscale, then just read it.
            leftImage=imread(leftImage);
        else
            error('The Color Type of Left Image is not acceptable. Acceptable
color types are truecolor or grayscale.');
        end
    end
catch
    % if it is not an image but a variable
    leftImage=leftImage;
end
try
    % Grab the image information (metadata) of right image using the function
imfinfo
    rightImageInfo=imfinfo(rightImage);
    % Since SHDR2L is applied on a grayscale image, determine if the
    % input right image is already in grayscale or color
    if(getfield(rightImageInfo,'ColorType')=='truecolor')
    % Read an image using imread function, convert from RGB color space to
    % grayscale using rgb2gray function and assign it to variable rightImage
        rightImage=rgb2gray(imread(rightImage));
    else if(getfield(rightImageInfo,'ColorType')=='grayscale')
    % If the image is already in grayscale, then just read it.
            rightImage=imread(rightImage);
        else
```

```
            error('The Color Type of Right Image is not acceptable.
Acceptable color types are truecolor or grayscale.');
        end
    end
catch
    % if it is not an image but a variable
    rightImage=rightImage;
end
% Find the size (columns and rows) of the left image and assign the rows to
% variable nrLeft, and columns to variable ncLeft
[nrLeft,ncLeft] = size(leftImage);
% Find the size (columns and rows) of the right image and assign the rows to
% variable nrRight, and columns to variable ncRight
[nrRight,ncRight] = size(rightImage);
% Check to see if both the left and right images have same number of rows
% and columns
if(nrLeft==nrRight && ncLeft==ncRight)
else
    error('Both left and right images should have the same number of rows and
columns');
end
% Check the size of window to see if it is an odd number.
if (mod(windowSize,2)==0)
    error('The window size must be an odd number.');
end
% Check whether minimum disparity is less than the maximum disparity.
if (dispMin>dispMax)
    error('Minimum Disparity must be less than the Maximum disparity.');
end
% Create an image of size nrLeft and ncLeft, fill it with zeros and assign
% it to variable dispMap
dispMap=zeros(nrLeft, ncLeft);
% Find out how many rows and columns are to the left/right/up/down of the
% central pixel based on the window size
win=(windowSize-1)/2;
numberOfBits=8;
tic; % Initialize the timer to calculate the time consumed.
for(i=1+win:1:nrLeft-win)
    for(j=1+win:1:ncLeft-win-dispMax)
        min=0;
        position=0;
        rightWindow=rightImage(i-win:i+win, j-win:j+win);
        for(dispRange=dispMin:1:dispMax)
            sad=0.0;
            if (j+win+dispRange <= ncLeft)
                leftWindow=leftImage(i-win:i+win,                          j-
win+dispRange:j+win+dispRange);
                bloc3=bitxor(rightWindow,leftWindow);
                distance=uint8(zeros(windowSize,windowSize));
                for (k=1:1:numberOfBits)
                    distance=distance+bitget(bloc3,k);
                end
                dif=sum(sum(distance));
                if (dispRange==0)
                    min=dif;
                elseif (min>dif)
                    min=dif;
```

```
                    position=dispRange;
                end
            end
        end
        dispMap(i,j) = position;
    end
end
% Stop the timer to calculate the time consumed.
timeTaken=toc;
```

A.19 Sum of Hamming Distances-Left to Right Matching

```
% ************************************************************************
% Title: Function-Compute Correlation between two images using the
% similarity measure of Sum of Hamming Distances (SHD) with Left Image
% as reference.
% Author: Siddhant Ahuja
% Created: May 2008
% Copyright Siddhant Ahuja, 2008
% Inputs: Left Image (var: leftImage), Right Image (var: rightImage),
% Window Size (var: windowSize), Minimum Disparity (dispMin), Maximum
% Disparity (dispMax)
% Outputs: Disparity Map (var: dispMap), Time taken (var: timeTaken)
%         Example         Usage         of         Function:         [dispMap,
timeTaken]=funcSHDL2R('TsukubaLeft.jpg', 'TsukubaRight.jpg', 9, 0, 16);
% ************************************************************************
function [dispMap, timeTaken]=funcSHDL2R(leftImage, rightImage, windowSize,
dispMin, dispMax)
try
    % Grab the image information (metadata) of left image using the function
imfinfo
    leftImageInfo=imfinfo(leftImage);
    % Since SHDL2R is applied on a grayscale image, determine if the
    % input left image is already in grayscale or color
    if(getfield(leftImageInfo,'ColorType')=='truecolor')
    % Read an image using imread function, convert from RGB color space to
    % grayscale using rgb2gray function and assign it to variable leftImage
        leftImage=rgb2gray(imread(leftImage));
    else if(getfield(leftImageInfo,'ColorType')=='grayscale')
    % If the image is already in grayscale, then just read it.
            leftImage=imread(leftImage);
        else
            error('The Color Type of Left Image is not acceptable. Acceptable
color types are truecolor or grayscale.');
        end
    end
catch
    % if it is not an image but a variable
    leftImage=leftImage;
end
try
    % Grab the image information (metadata) of right image using the function
imfinfo
```

```
    rightImageInfo=imfinfo(rightImage);
    % Since SHDL2R is applied on a grayscale image, determine if the
    % input right image is already in grayscale or color
    if(getfield(rightImageInfo,'ColorType')=='truecolor')
    % Read an image using imread function, convert from RGB color space to
    % grayscale using rgb2gray function and assign it to variable rightImage
        rightImage=rgb2gray(imread(rightImage));
    else if(getfield(rightImageInfo,'ColorType')=='grayscale')
    % If the image is already in grayscale, then just read it.
            rightImage=imread(rightImage);
        else
            error('The Color Type of Right Image is not acceptable.
Acceptable color types are truecolor or grayscale.');
        end
    end
catch
    % if it is not an image but a variable
    rightImage=rightImage;
end
% Find the size (columns and rows) of the left image and assign the rows to
% variable nrLeft, and columns to variable ncLeft
[nrLeft,ncLeft] = size(leftImage);
% Find the size (columns and rows) of the right image and assign the rows to
% variable nrRight, and columns to variable ncRight
[nrRight,ncRight] = size(rightImage);
% Check to see if both the left and right images have same number of rows
% and columns
if(nrLeft==nrRight && ncLeft==ncRight)
else
    error('Both left and right images should have the same number of rows and
columns');
end
% Check the size of window to see if it is an odd number.
if (mod(windowSize,2)==0)
    error('The window size must be an odd number.');
end
% Check whether minimum disparity is less than the maximum disparity.
if (dispMin>dispMax)
    error('Minimum Disparity must be less than the Maximum disparity.');
end
% Create an image of size nrLeft and ncLeft, fill it with zeros and assign
% it to variable dispMap
dispMap=zeros(nrLeft, ncLeft);
% Find out how many rows and columns are to the left/right/up/down of the
% central pixel based on the window size
win=(windowSize-1)/2;
numberOfBits=8;
tic; % Initialize the timer to calculate the time consumed.
for(i=1+win:1:nrLeft-win)
    for(j=1+win+dispMax:1:ncLeft-win)
        min=0;
        position=0;
        leftWindow=leftImage(i-win:i+win, j-win:j+win);
        for(dispRange=-dispMin:-1:-dispMax)
            if (j-win+dispRange > 0)
                rightWindow=rightImage(i-win:i+win,                    j-
win+dispRange:j+win+dispRange);
```

```
                bloc3=bitxor(leftWindow,rightWindow);
                distance=uint8(zeros(windowSize,windowSize));
                for (k=1:1:numberOfBits)
                    distance=distance+bitget(bloc3,k);
                end
                dif=sum(sum(distance));
                if (dispRange==0)
                    min=dif;
                elseif (min>dif)
                    min=dif;
                    position=dispRange;
                end
            end
        end
        dispMap(i,j) = -position;
    end
end
% Stop the timer to calculate the time consumed.
timeTaken=toc;
```

A.20 Finding Occluded Regions

```
% ************************************************************************
% Title: Function-Find Occluded regions of an image
% Notes: Occluded regions are defined as regions that are occluded in the
% matching image, i.e., where the forward-mapped disparity
% lands at a location with a larger (nearer) disparity.
% Author: Siddhant Ahuja
% Created: September 2008
% Copyright Siddhant Ahuja, 2008
% Inputs: Left to Right Disparity Image (var: dispMapL2R), Right to Left
Disparity Image (var: dispMapR2L),
% Scale factor of ground truth map (var: scale) typically 4.0, Threshold for
the check (var: thresh) typically 2.0
% Outputs: Disparity Map (var: dispMap), Time taken (var: timeTaken)
%       Example       Usage       of       Function:       [occludedImg,
timeTaken]=funcOccludedRegions('disp_1_r.png', 'disp_r_1.png', 4, 2);
% ************************************************************************
function [occludedImg, timeTaken]=funcOccludedRegions(dispMapL2R, dispMapR2L,
scale, thresh)
% Initiate the Timer to calculate the time consumed.
tic;
dispMapL2R=imread(dispMapL2R);
dispMapR2L=imread(dispMapR2L);
dispMapL2R = floor(double (dispMapL2R)/scale);
dispMapR2L = floor(double (dispMapR2L)/scale);
% Prepare matrix for subtraction and scale it for comparison
dispMapL2R=-dispMapL2R;
% Find the size (columns and rows) of the L2R Disparity map and assign the
rows to
% variable nrLRCCheck, and columns to variable ncLRCCheck
[nrLRCCheck,ncLRCCheck] = size(dispMapL2R);
```

```
% Create an image of size nrLRCCheck and ncLRCCheck, fill it with zeros and
assign
% it to variable occludedImg
occludedImg=zeros(nrLRCCheck,ncLRCCheck);
for(i=1:1:nrLRCCheck)
    for(j=1:1:ncLRCCheck)
        xl=j;
        xr=xl+dispMapL2R(i,xl);
        if (xr>ncLRCCheck||xr<1)
            occludedImg(i,j) = 0; %% occluded pixel
        else
            xlp=xr+dispMapR2L(i,xr);
            if (abs(xl-xlp)<thresh)
                occludedImg(i,j) = -dispMapL2R(i,j);  %% non-occluded pixel
            else
                occludedImg(i,j) = 0; %% occluded pixel
            end
        end
    end
end
% Terminate the Timer to calculate the time consumed.
timeTaken=toc;
```

A.21 Finding Depth-Discontinuous Regions

```
% ************************************************************************
% Title: Function-Find Discontinuous regions of an image
% Notes:
% 1. According to paper [A Taxonomy and Evaluation of Dense Two-Frame Stereo
Correspondence Algorithms], Depth Discontinuous regions are defined as pixels
whose neighboring
% disparities differ by more than a certain gap, dilated
% by a window of width windowSize.
% 2. According to paper [ An Experimental Comparison of Stereo Algorithms], A
pixel is a depth
% discontinuity if any of its (4-connected) neighbors has a disparity that
differs by more than 1 from
% its disparity. Neighboring pixels that are part of a sloped surface can
easily differ by 1 pixel, but
% should not be counted as discontinuities.
% Author: Siddhant Ahuja
% Created: May 2008
% Copyright Siddhant Ahuja, 2008
%  Inputs: Input Image-Depth Map (var: inputImage), Window Size (var:
windowSize),
% Threshold (var: thresh) Typical value is 3
%  Outputs: Discontinuous Map (var: discontinuousImg) , Time taken (var:
timeTaken)
%       Example       Usage       of       Function:       [discontinuousImg,
timeTaken]=funcDiscontinuousRegions('TsukubaGroundTruthL2R.pgm', 9, 3);
% ************************************************************************
```

```
function [discontinuousImg, timeTaken]=funcDiscontinuousRegions(inputImage,
windowSize, thresh)
% Read the input image
try
    % Read an image using imread function
    inputImage=imread(inputImage);
    % grab the number of rows, columns, and channels
    [nr, nc, nChannels]=size(inputImage);
     % Grab the image information (metadata) of input image using the
function imfinfo
    inputImageInfo=imfinfo(inputImage);
    % Determine if input left image is already in grayscale or color
    if(getfield(inputImageInfo,'ColorType')=='truecolor')
        inputImage=rgb2gray(inputImage);
    else if(getfield(inputImageInfo,'ColorType')=='grayscale')
        inputImage=inputImage;
        else
        error('The Color Type of Left Image is not acceptable. Acceptable
color types are truecolor or grayscale.');
        end
    end
catch
    % if it is not an image but a variable
    % grab the number of channels
    [nr, nc, nChannels]=size(inputImage);
    if(nChannels)>1
        inputImage=rgb2gray(inputImage);
    else
        inputImage=inputImage;
    end
end
% Check the size of window to see if it is an odd number.
if (mod(windowSize,2)==0)
    error('The window size must be an odd number.');
end
% Create an image of size nr and nc, fill it with zeros and assign
% it to variable discontinuousImg
discontinuousImg=zeros(nr, nc);
% Create an image of size nr and nc, fill it with zeros and assign
% it to variable edgeImg
edgeImg=zeros(nr, nc);
% Find out how many rows and columns are to the left/right/up/down of the
% central pixel based on the window size
win=(windowSize-1)/2;
inputImage=double(inputImage);
tic; % Initialize the timer to calculate the time consumed.
% Find variation/edges in disparity values in horizontal and vertical
directions
for (i=1:1:nr-1)
    for (j=1:1:nc-1)
        % Traverse in horizontal direction
        if (abs(inputImage(i,j)-inputImage(i,j+1)) > thresh )
            edgeImg(i,j) = 255;
            edgeImg(i,j+1) = 255;
        end
        % Traverse in vertical direction
        if (abs(inputImage(i,j)-inputImage(i+1,j)) > thresh )
```

```
            edgeImg(i,j) = 255;
            edgeImg(i+1,j) = 255;
        end
    end
end
% Dilate within the window
for (i=1+win:nr-win)
    for (j=1+win:nc-win)
        % Go over the square window
        sum = 0.0;
        for (a=-win:1:win)
            for (b=-win:1:win)
                sum = sum + edgeImg(i+a,j+b);
            end
        end
        % Apply Threshold
        if (sum>0)
            discontinuousImg(i,j) = 255;
        else
            discontinuousImg(i,j) = 0;
        end
    end
end
% Stop the timer to calculate the time consumed.
timeTaken=toc;
```

A.22 Finding Low-Texture Regions

```
% ************************************************************************
% Title: Function-Find Textureless regions of an image
% Notes: Textureless regions are defined as regions where the squared
horizontal
% intensity gradient averaged over a square window of a given size
% (windowSize) is below a given threshold (thresh);
% Author: Siddhant Ahuja
% Created: May 2008
% Copyright Siddhant Ahuja, 2008
% Inputs: Input Image (var: inputImage), Window Size (var: windowSize),
% Threshold (var: thresh) Typical value is 4
% Outputs: Textureless Map (var: texturelessImg) , Time taken (var:
timeTaken)
% Example Usage of Function: [texturelessImg,
timeTaken]=funcTexturelessRegions('TsukubaLeftColor.png', 9, 4);
% ************************************************************************
function [texturelessImg, timeTaken]=funcTexturelessRegions(inputImage,
windowSize, thresh)
% Read the input image
try
    % Read an image using imread function
    inputImage=imread(inputImage);
    % grab the number of rows, columns, and channels
    [nr, nc, nChannels]=size(inputImage);
```

```
    % Grab the image information (metadata) of input image using the
function imfinfo
    inputImageInfo=imfinfo(inputImage);
    % Determine if input left image is already in grayscale or color
    if(getfield(inputImageInfo,'ColorType')=='truecolor')
        colored=1;
    else if(getfield(inputImageInfo,'ColorType')=='grayscale')
        colored=0;
        else
        error('The Color Type of Left Image is not acceptable. Acceptable
color types are truecolor or grayscale.');
        end
    end
catch
    % if it is not an image but a variable
    % grab the number of channels
    [nr, nc, nChannels]=size(inputImage);
    if(nChannels)>1
        colored=1;
    else
        colored=0;
    end
end
% Check the size of window to see if it is an odd number.
if (mod(windowSize,2)==0)
    error('The window size must be an odd number.');
end
% Create an image of size nr and nc, fill it with zeros and assign
% it to variable texturelessImg
texturelessImg=zeros(nr, nc);
% Create an image of size nr and nc, fill it with zeros and assign
% it to variable sqGradImg
sqGradImg=zeros(nr,nc);
% Find out how many rows and columns are to the left/right/up/down of the
% central pixel based on the window size
win=(windowSize-1)/2;
inputImage=double(inputImage);
tic; % Initialize the timer to calculate the time consumed.
% Produce Squared Horizontal Gradient image sqGradImg
for (i=1:1:nr)
    for (j=1:1:nc-1)
        sum = 0.0;
        for (k=1:1:nChannels)
            diff =  inputImage(i,j,k) - inputImage(i,j+1,k);
            sum = sum + (diff*diff);
        end
        sum = sum / nChannels;
        sqGradImg(i,j+1) = sum;
        if (j==1)
            sqGradImg(i,j) = sum;
        end
        if (sum > sqGradImg(i,j))
            sqGradImg(i,j) = sum;
        end
    end
end
% Compute average within predefined box window of size windowSize x
```

```
% windowSize
for (i=1+win:nr-win)
    for (j=1+win:nc-win)
        % go over the square window
        sum = 0.0;
        avg = 0.0;
        for (a=-win:1:win)
            for (b=-win:1:win)
                sum = sum + sqGradImg(i+a,j+b);
            end
        end
        % Compute the average
        avg = sum / (windowSize*windowSize);
        % Apply threshold
        if (avg < (thresh*thresh))
            texturelessImg(i,j) = 255; % mark detected textureless pixel as
white
        end
    end
end
% Stop the timer to calculate the time consumed.
timeTaken=toc;
```

A.23 Introduce Vignetting effect

```
% ************************************************************************
% Title: Function-Introduce Vignetting effect to the image
% Author: Siddhant Ahuja
% Created: September 2008
% Copyright Siddhant Ahuja, 2008
% Inputs: Input Image (var: inputImage), Scale Change level (var:
% scaleLevel) Valid values for scale change should go from 0.1 to 1
% Outputs: Vignetting effect added image (var: vignettImg, Time taken (var:
timeTaken)
%        Example        Usage        of        Function:        [vignettImg,
timeTaken]=funcVignettingEffect('TsukubaLeftColor.png', 0.5);
% ************************************************************************
function [vignettImg, timeTaken]=funcVignettingEffect(inputImage,scaleLevel)
% Read the input image
try
    % Read an image using imread function
    inputImage=imread(inputImage);
    % grab the number of rows, columns, and channels
    [nr, nc, nChannels]=size(inputImage);
     % Grab the image information (metadata) of input image using the
function imfinfo
    inputImageInfo=imfinfo(inputImage);
    % Determine if input left image is already in grayscale or color
    if(getfield(inputImageInfo,'ColorType')=='truecolor')
        colored=1;
    else if(getfield(leftImageInfo,'ColorType')=='grayscale')
        colored=0;
        else
```

```
        error('The Color Type of Left Image is not acceptable. Acceptable
color types are truecolor or grayscale.');
        end
    end
catch
    % if it is not an image but a variable
    inputImage=inputImage;
    % grab the number of channels
    [nr, nc, nChannels]=size(inputImage);
    if(nChannels)>1
        colored=1;
    else
        colored=0;
    end
end
if scaleLevel <= 0
    error('Scale value must be > 0');
end
vignettImg=inputImage;
tic; % Initialize the timer to calculate the time consumed.
imgCntX = nc/2;
imgCntY = nr/2;
maxDistance = sqrt (imgCntY^2 + imgCntX^2);
if(colored==1)
    for (i=1:nr)
        for (j=1:nc)
           dis = sqrt (abs(i-imgCntY)^2 + abs(j-imgCntX)^2);
           %% reduce brighness of pixel based on distance from the image
center
           vignettImg(i,j,1)     =     vignettImg(i,j,1)*     (1     -     (1-
scaleLevel)*(dis/maxDistance) );
           vignettImg(i,j,2)     =     vignettImg(i,j,2)*     (1     -     (1-
scaleLevel)*(dis/maxDistance) );
           vignettImg(i,j,3)     =     vignettImg(i,j,3)*     (1     -     (1-
scaleLevel)*(dis/maxDistance) );
       end
    end
else
%gray
     for (i=1:nr)
        for (j=1:nc)
           dis = sqrt (abs(i-imgCntY)^2 + abs(j-imgCntX)^2);
           %% reduce brighness of pixel based on distance from the image
center
           vignettImg(i,j)      =      vignettImg(i,j)*      (1      -      (1-
scaleLevel)*(dis/maxDistance) );
       end
    end
end
% Stop the timer to calculate the time consumed.
timeTaken=toc;
```

A.24 Introduce Scale change

```
% ************************************************************************
% Title: Function-Add scale change to the image.
% Author: Siddhant Ahuja
% Created: September 2008
% Copyright Siddhant Ahuja, 2008
% Inputs: Input Image (var: inputImage), Scale Change level (var:
% scaleLevel)Valid values for scale change should go from 0.1 to 1
% Outputs: Scale changed image (var: scaledImg, Time taken (var: timeTaken)
%         Example         Usage         of         Function:         [scaledImg,
timeTaken]=funcScaleChange('TsukubaLeftColor.png', 0.5);
% ************************************************************************
function [scaledImg, timeTaken]=funcScaleChange(inputImage,scaleLevel)
% Read the input image
try
    inputImage=imread(inputImage);
catch
    inputImage=inputImage;
end
tic; % Initialize the timer to calculate the time consumed.
scaledImg = inputImage*(scaleLevel*10);
% Stop the timer to calculate the time consumed.
timeTaken=toc;
```

A.25 Introduce Gaussian noise

```
% ************************************************************************
% Title: Function-Add gaussian noise to the image.
% Notes: Gaussian white noise has mean of 0 and
% variance V as the noiseLevel. When unspecified, M and V default to 0 and
0.01 respectively.
% The mean and variance parameters for 'gaussian' noise are always specified
as if for a double image
% in the range [0, 1].  If the input image is of class uint8 or uint16,
% the imnoise function converts the image to double, adds noise
% according to the specified type and parameters, and then converts the
% noisy image back to the same class as the input.
% Author: Siddhant Ahuja
% Created: September 2008
% Copyright Siddhant Ahuja, 2008
%  Inputs:  Input  Image  (var:  inputImage),  Noise  level  (var:  noiseLevel)
corresponds                                to                                15dB;
%0.00002=>50;0.00004=>47;0.00006=>45;0.000125=>42;0.00020=>40;0.00035=>37.5;0
.0006=>35;0.00117=>32.5;0.00195=>30;0.0037=>27.5;0.0065=>25;0.012=>22.5;0.021
=>20;0.043=>17.5;
%0.09=>15;0.27=>12.5
% Outputs: Gaussian noise added image (var: gaussNoiseImg, Time taken (var:
timeTaken)
```

```
%       Example       Usage       of       Function:       [gaussNoiseImg,
timeTaken]=funcGaussianNoise('TsukubaLeft.jpg', 0.09);
% ************************************************************************
function [gaussNoiseImg, timeTaken]=funcGaussianNoise(inputImage,noiseLevel)
% Read the input image
try
    inputImage=imread(inputImage);
catch
    inputImage=inputImage;
end
tic; % Initialize the timer to calculate the time consumed.
gaussNoiseImg = imnoise (inputImage, 'gaussian', 0,noiseLevel);
% Stop the timer to calculate the time consumed.
timeTaken=toc;
```

A.26 Adjust Gamma variation

```
% ************************************************************************
% Title: Function-Add gamma change to the image.
% Author: Siddhant Ahuja
% Created: September 2008
% Copyright Siddhant Ahuja, 2008
% Inputs: Input Image (var: inputImage), Gamma Change level (var:
% gammaLevel)Valid values for gamma change should go from 0.1 to 5
% Outputs: Gamma changed image (var: gammaImg, Time taken (var: timeTaken)
%         Example         Usage         of         Function:         [gammaImg,
timeTaken]=funcGammaChange('TsukubaLeftColor.png', 0.5);
% ************************************************************************
function [gammaImg, timeTaken]=funcGammaChange(inputImage,gammaLevel)
% Read the input image
try
    % Read an image using imread function
    inputImage=imread(inputImage);
     % Grab the image information (metadata) of input image using the
function imfinfo
    inputImageInfo=imfinfo(inputImage);
    % Determine if input left image is already in grayscale or color
    if(getfield(inputImageInfo,'ColorType')=='truecolor')
        colored=1;
    else if(getfield(leftImageInfo,'ColorType')=='grayscale')
        colored=0;
        else
        error('The Color Type of Left Image is not acceptable. Acceptable
color types are truecolor or grayscale.');
        end
    end
catch
    % if it is not an image but a variable
    inputImage=inputImage;
    % grab the number of channels
    [nr, nc, nChannels]=size(inputImage);
    if(nChannels)>1
```

```
        colored=1;
    else
        colored=0;
    end
end
if gammaLevel <= 0
    error('Gamma value must be > 0');
end
if isa(inputImage,'uint8');
    gammaImg = double(inputImage);
else
    gammaImg = inputImage;
end
tic; % Initialize the timer to calculate the time consumed.
if(colored==1)
    % Red component
    gammaImg(:,:,1) = gammaImg(:,:,1)-min(min(gammaImg(:,:,1)));
    gammaImg(:,:,1) = gammaImg(:,:,1)./max(max(gammaImg(:,:,1)));
    gammaImg(:,:,1)  =    gammaImg(:,:,1).^(1/gammaLevel);      % Apply  gamma
function
    % Green component
    gammaImg(:,:,2) = gammaImg(:,:,2)-min(min(gammaImg(:,:,2)));
    gammaImg(:,:,2) = gammaImg(:,:,2)./max(max(gammaImg(:,:,2)));
    gammaImg(:,:,2)  =    gammaImg(:,:,2).^(1/gammaLevel);      % Apply  gamma
function
    % Blue component
    gammaImg(:,:,3) = gammaImg(:,:,3)-min(min(gammaImg(:,:,3)));
    gammaImg(:,:,3) = gammaImg(:,:,3)./max(max(gammaImg(:,:,3)));
    gammaImg(:,:,3)  =    gammaImg(:,:,3).^(1/gammaLevel);      % Apply  gamma
function
else
    gammaImg(:,:) = gammaImg(:,:)-min(min(gammaImg(:,:)));
    gammaImg(:,:) = gammaImg(:,:)./max(max(gammaImg(:,:)));
    gammaImg(:,:) =  gammaImg(:,:).^(1/gammaLevel);   % Apply gamma function
end
% Stop the timer to calculate the time consumed.
timeTaken=toc;
```

A.27 Generate Variance Map

```
% ************************************************************************
% Title: Function-Compute Variance map of the image
% Author: Siddhant Ahuja
% Created: May 2008
% Copyright Siddhant Ahuja, 2008
% Inputs: Input Image (var: inputImage), Window Size (var: windowSize),
% Threshold (var: thresh) Typical value is 140
% Outputs: Variance Map (var: varianceImg) , Time taken (var: timeTaken)
%        Example        Usage        of        Function:        [varianceImg,
timeTaken]=funcVarianceMap('TsukubaLeftColor.png', 9, 1);
% ************************************************************************
```

```
function [varianceImg, timeTaken]=funcVarianceMap(inputImage, windowSize,
thresh)
try
    % Grab the image information (metadata) of input image using the function
imfinfo
    inputImageInfo=imfinfo(inputImage);
    if(getfield(inputImageInfo,'ColorType')=='truecolor')
    % Read an image using imread function, convert from RGB color space to
    % grayscale using rgb2gray function and assign it to variable inputImage
        inputImage=rgb2gray(imread(inputImage));
        % Convert the image from uint8 to double
        inputImage=double(inputImage);
    else if(getfield(inputImageInfo,'ColorType')=='grayscale')
    % If the image is already in grayscale, then just read it.
            inputImage=imread(inputImage);
            % Convert the image from uint8 to double
            inputImage=double(inputImage);
        else
            error('The Color Type of Left Image is not acceptable. Acceptable
color types are truecolor or grayscale.');
        end
    end
catch
    % if it is not an image but a variable
    inputImage=inputImage;
end
% Find the size (columns and rows) of the input image and assign the rows to
% variable nr, and columns to variable nc
[nr,nc] = size(inputImage);
% Check the size of window to see if it is an odd number.
if (mod(windowSize,2)==0)
    error('The window size must be an odd number.');
end
% Create an image of size nr and nc, fill it with zeros and assign
% it to variable meanImg
meanImg=zeros(nr, nc);
% Create an image of size nr and nc, fill it with zeros and assign
% it to variable varianceImg
varianceImg=zeros(nr, nc);
% Find out how many rows and columns are to the left/right/up/down of the
% central pixel based on the window size
win=(windowSize-1)/2;
tic; % Initialize the timer to calculate the time consumed.
% Compute a map of mean values
for(i=1+win:1:nr-win)
    for(j=1+win:1:nc-win)
        sum=0.0;
        for(a=-win:1:win)
            for(b=-win:1:win)
                sum=sum+inputImage(i+a,j+b);
            end
        end
        meanImg(i,j)=sum/(windowSize*windowSize);
    end
end
% Compute a map of variance values
for(i=1+win:1:nr-win)
```

```
    for(j=1+win:1:nc-win)
        sum=0.0;
        for(a=-win:1:win)
            for(b=-win:1:win)
                sum=sum+((inputImage(i+a,j+b)-meanImg(i,j))^2);
            end
        end
        varianceImg(i,j)=sum/((windowSize*windowSize)-1);
    end
end
% Apply threshold to produce a binarized variance map
for(i=1+win:1:nr-win)
    for(j=1+win:1:nc-win)
        if (varianceImg(i,j) > thresh)
            varianceImg(i,j) = 255;
        else
            varianceImg(i,j) = 0;
        end
    end
end
% Stop the timer to calculate the time consumed.
timeTaken=toc;
```

A.28 Post-processing algorithm

```
% ************************************************************************
% Title: Function-Post process an image
% Author: Siddhant Ahuja
% Created: Septembed 2008
% Copyright Siddhant Ahuja, 2008
% Inputs: Image (var: inputImage), Disparity Map (var: dispMap), Threshold
% for variance map of input image (var: threshTxt) typically 135, Threshold
for variance
% map of input Disparity Map (var: threshTdisp) typically 17
% Outputs: Post Processed Image (var: postProcessedImg),
% Time taken (var: timeTaken)
% Example Usage of Function: [postProcessedImg,
% timeTaken]=funcPostProcess('imL.png','disp_1_r.png', 135, 17)
% ************************************************************************
function [postProcessedImg, timeTaken]=funcPostProcess(inputImage, dispMap,
threshTxt, threshTdisp)
tic; % Initialize the timer to calculate the time consumed.
% Compute Variance map of input image
[varianceInputImg, time]=funcVarianceMap(inputImage, 9, threshTxt);
% Compute Variance map of input disparity map
[varianceDispMap, time]=funcVarianceMap(dispMap, 9, threshTxt);
postProcessedImg=imread(dispMap);
[nr, nc]=size(postProcessedImg);
for(i=1:1:nr)
    for(j=1:1:nc)
        if(varianceInputImg(i,j)<=threshTxt                                    ||
varianceDispMap(i,j)>=threshTdisp)
```

```
            postProcessedImg(i,j)=0;
        end
    end
end
% Stop the timer to calculate the time consumed.
timeTaken=toc;
```

A.29 Initialize 2001 Dataset from Middlebury

```
% Initialize 2001 Dataset array of structures

% barn1
Dataset2001(1).index=1;
Dataset2001(1).name='barn1';
Dataset2001(1).leftImage='im2.ppm';
Dataset2001(1).rightImage='im6.ppm';
Dataset2001(1).groundTruthL2R='disp2.pgm';
Dataset2001(1).groundTruthR2L='disp6.pgm';
Dataset2001(1).dispMax=16;
Dataset2001(1).scale=8;

% barn2
Dataset2001(2).index=2;
Dataset2001(2).name='barn2';
Dataset2001(2).leftImage='im2.ppm';
Dataset2001(2).rightImage='im6.ppm';
Dataset2001(2).groundTruthL2R='disp2.pgm';
Dataset2001(2).groundTruthR2L='disp6.pgm';
Dataset2001(2).dispMax=20;
Dataset2001(2).scale=8;

% bull
Dataset2001(3).index=3;
Dataset2001(3).name='bull';
Dataset2001(3).leftImage='im2.ppm';
Dataset2001(3).rightImage='im6.ppm';
Dataset2001(3).groundTruthL2R='disp2.pgm';
Dataset2001(3).groundTruthR2L='disp6.pgm';
Dataset2001(3).dispMax=20;
Dataset2001(3).scale=8;

% map
Dataset2001(4).index=4;
Dataset2001(4).name='map';
Dataset2001(4).leftImage='im0.pgm';
Dataset2001(4).rightImage='im1.pgm';
Dataset2001(4).groundTruthL2R='disp0.pgm';
Dataset2001(4).groundTruthR2L='disp1.pgm';
Dataset2001(4).dispMax=30;
Dataset2001(4).scale=8;

% poster
Dataset2001(5).index=5;
```

```
Dataset2001(5).name='poster';
Dataset2001(5).leftImage='im2.ppm';
Dataset2001(5).rightImage='im6.ppm';
Dataset2001(5).groundTruthL2R='disp2.pgm';
Dataset2001(5).groundTruthR2L='disp6.pgm';
Dataset2001(5).dispMax=20;
Dataset2001(5).scale=8;

% sawtooth
Dataset2001(6).index=6;
Dataset2001(6).name='sawtooth';
Dataset2001(6).leftImage='im2.ppm';
Dataset2001(6).rightImage='im6.ppm';
Dataset2001(6).groundTruthL2R='disp2.pgm';
Dataset2001(6).groundTruthR2L='disp6.pgm';
Dataset2001(6).dispMax=20;
Dataset2001(6).scale=8;

% tsukuba
Dataset2001(7).index=7;
Dataset2001(7).name='tsukuba';
Dataset2001(7).leftImage='imL.png';
Dataset2001(7).rightImage='imR.png';
Dataset2001(7).groundTruthL2R='truedisp1.row3.col3.pgm';
Dataset2001(7).groundTruthR2L='truedisp2.row3.col3.pgm';
Dataset2001(7).dispMax=30;
Dataset2001(7).scale=16;

% venus
Dataset2001(8).index=8;
Dataset2001(8).name='venus';
Dataset2001(8).leftImage='imL.png';
Dataset2001(8).rightImage='imR.png';
Dataset2001(8).groundTruthL2R='disp2.pgm';
Dataset2001(8).groundTruthR2L='disp6.pgm';
Dataset2001(8).dispMax=20;
Dataset2001(8).scale=8;
```

A.30 Initialize 2003 Dataset from Middlebury

```
% Initialize 2003 Dataset array of structures

% teddy
Dataset2003(1).index=1;
Dataset2003(1).name='teddy';
Dataset2003(1).leftImage='imL.png';
Dataset2003(1).rightImage='imR.png';
Dataset2003(1).groundTruthL2R='disp2.png';
Dataset2003(1).groundTruthR2L='disp6.png';
Dataset2003(1).dispMax=53;
Dataset2003(1).scale=4;
```

```
% cones
Dataset2003(2).index=2;
Dataset2003(2).name='cones';
Dataset2003(2).leftImage='imL.png';
Dataset2003(2).rightImage='imR.png';
Dataset2003(2).groundTruthL2R='disp2.png';
Dataset2003(2).groundTruthR2L='disp6.png';
Dataset2003(2).dispMax=55;
Dataset2003(2).scale=4;
```

A.31 Initialize 2006 Dataset from Middlebury

```
% Initialize 2006 Dataset array of structures

% aloe
Dataset2006(1).index=1;
Dataset2006(1).name='aloe';
Dataset2006(1).leftImage='view1.png';
Dataset2006(1).rightImage='view5.png';
Dataset2006(1).groundTruthL2R='disp1.png';
Dataset2006(1).groundTruthR2L='disp5.png';
Dataset2006(1).dispMax=70;
Dataset2006(1).scale=3;

% baby1
Dataset2006(2).index=2;
Dataset2006(2).name='baby1';
Dataset2006(2).leftImage='view1.png';
Dataset2006(2).rightImage='view5.png';
Dataset2006(2).groundTruthL2R='disp1.png';
Dataset2006(2).groundTruthR2L='disp5.png';
Dataset2006(2).dispMax=45;
Dataset2006(2).scale=3;

% baby2
Dataset2006(3).index=3;
Dataset2006(3).name='baby2';
Dataset2006(3).leftImage='view1.png';
Dataset2006(3).rightImage='view5.png';
Dataset2006(3).groundTruthL2R='disp1.png';
Dataset2006(3).groundTruthR2L='disp5.png';
Dataset2006(3).dispMax=53;
Dataset2006(3).scale=3;

% baby3
Dataset2006(4).index=4;
Dataset2006(4).name='baby3';
Dataset2006(4).leftImage='view1.png';
Dataset2006(4).rightImage='view5.png';
Dataset2006(4).groundTruthL2R='disp1.png';
Dataset2006(4).groundTruthR2L='disp5.png';
```

```
Dataset2006(4).dispMax=51;
Dataset2006(4).scale=3;

% bowling1
Dataset2006(5).index=5;
Dataset2006(5).name='bowling1';
Dataset2006(5).leftImage='view1.png';
Dataset2006(5).rightImage='view5.png';
Dataset2006(5).groundTruthL2R='disp1.png';
Dataset2006(5).groundTruthR2L='disp5.png';
Dataset2006(5).dispMax=77;
Dataset2006(5).scale=3;

% bowling2
Dataset2006(6).index=6;
Dataset2006(6).name='bowling2';
Dataset2006(6).leftImage='view1.png';
Dataset2006(6).rightImage='view5.png';
Dataset2006(6).groundTruthL2R='disp1.png';
Dataset2006(6).groundTruthR2L='disp5.png';
Dataset2006(6).dispMax=67;
Dataset2006(6).scale=3;

% cloth1
Dataset2006(7).index=7;
Dataset2006(7).name='cloth1';
Dataset2006(7).leftImage='view1.png';
Dataset2006(7).rightImage='view5.png';
Dataset2006(7).groundTruthL2R='disp1.png';
Dataset2006(7).groundTruthR2L='disp5.png';
Dataset2006(7).dispMax=57;
Dataset2006(7).scale=3;

% cloth2
Dataset2006(8).index=8;
Dataset2006(8).name='cloth2';
Dataset2006(8).leftImage='view1.png';
Dataset2006(8).rightImage='view5.png';
Dataset2006(8).groundTruthL2R='disp1.png';
Dataset2006(8).groundTruthR2L='disp5.png';
Dataset2006(8).dispMax=77;
Dataset2006(8).scale=3;

% cloth3
Dataset2006(9).index=9;
Dataset2006(9).name='cloth3';
Dataset2006(9).leftImage='view1.png';
Dataset2006(9).rightImage='view5.png';
Dataset2006(9).groundTruthL2R='disp1.png';
Dataset2006(9).groundTruthR2L='disp5.png';
Dataset2006(9).dispMax=55;
Dataset2006(9).scale=3;

% cloth4
Dataset2006(10).index=10;
Dataset2006(10).name='cloth4';
```

```
Dataset2006(10).leftImage='view1.png';
Dataset2006(10).rightImage='view5.png';
Dataset2006(10).groundTruthL2R='disp1.png';
Dataset2006(10).groundTruthR2L='disp5.png';
Dataset2006(10).dispMax=67;
Dataset2006(10).scale=3;

% flowerpots
Dataset2006(11).index=11;
Dataset2006(11).name='flowerpots';
Dataset2006(11).leftImage='view1.png';
Dataset2006(11).rightImage='view5.png';
Dataset2006(11).groundTruthL2R='disp1.png';
Dataset2006(11).groundTruthR2L='disp5.png';
Dataset2006(11).dispMax=61;
Dataset2006(11).scale=3;

% lampshade1
Dataset2006(12).index=12;
Dataset2006(12).name='lampshade1';
Dataset2006(12).leftImage='view1.png';
Dataset2006(12).rightImage='view5.png';
Dataset2006(12).groundTruthL2R='disp1.png';
Dataset2006(12).groundTruthR2L='disp5.png';
Dataset2006(12).dispMax=65;
Dataset2006(12).scale=3;

% lampshade2
Dataset2006(13).index=13;
Dataset2006(13).name='lampshade2';
Dataset2006(13).leftImage='view1.png';
Dataset2006(13).rightImage='view5.png';
Dataset2006(13).groundTruthL2R='disp1.png';
Dataset2006(13).groundTruthR2L='disp5.png';
Dataset2006(13).dispMax=65;
Dataset2006(13).scale=3;

% midd1
Dataset2006(14).index=14;
Dataset2006(14).name='midd1';
Dataset2006(14).leftImage='view1.png';
Dataset2006(14).rightImage='view5.png';
Dataset2006(14).groundTruthL2R='disp1.png';
Dataset2006(14).groundTruthR2L='disp5.png';
Dataset2006(14).dispMax=69;
Dataset2006(14).scale=3;

% midd2
Dataset2006(15).index=15;
Dataset2006(15).name='midd2';
Dataset2006(15).leftImage='view1.png';
Dataset2006(15).rightImage='view5.png';
Dataset2006(15).groundTruthL2R='disp1.png';
Dataset2006(15).groundTruthR2L='disp5.png';
Dataset2006(15).dispMax=63;
Dataset2006(15).scale=3;
```

```
% monopoly
Dataset2006(16).index=16;
Dataset2006(16).name='monopoly';
Dataset2006(16).leftImage='view1.png';
Dataset2006(16).rightImage='view5.png';
Dataset2006(16).groundTruthL2R='disp1.png';
Dataset2006(16).groundTruthR2L='disp5.png';
Dataset2006(16).dispMax=65;
Dataset2006(16).scale=3;

% plastic
Dataset2006(17).index=17;
Dataset2006(17).name='plastic';
Dataset2006(17).leftImage='view1.png';
Dataset2006(17).rightImage='view5.png';
Dataset2006(17).groundTruthL2R='disp1.png';
Dataset2006(17).groundTruthR2L='disp5.png';
Dataset2006(17).dispMax=65;
Dataset2006(17).scale=3;

% rocks1
Dataset2006(18).index=18;
Dataset2006(18).name='rocks1';
Dataset2006(18).leftImage='view1.png';
Dataset2006(18).rightImage='view5.png';
Dataset2006(18).groundTruthL2R='disp1.png';
Dataset2006(18).groundTruthR2L='disp5.png';
Dataset2006(18).dispMax=57;
Dataset2006(18).scale=3;

% rocks2
Dataset2006(19).index=19;
Dataset2006(19).name='rocks2';
Dataset2006(19).leftImage='view1.png';
Dataset2006(19).rightImage='view5.png';
Dataset2006(19).groundTruthL2R='disp1.png';
Dataset2006(19).groundTruthR2L='disp5.png';
Dataset2006(19).dispMax=57;
Dataset2006(19).scale=3;

% wood1
Dataset2006(20).index=20;
Dataset2006(20).name='wood1';
Dataset2006(20).leftImage='view1.png';
Dataset2006(20).rightImage='view5.png';
Dataset2006(20).groundTruthL2R='disp1.png';
Dataset2006(20).groundTruthR2L='disp5.png';
Dataset2006(20).dispMax=73;
Dataset2006(20).scale=3;

% wood2
Dataset2006(21).index=21;
Dataset2006(21).name='wood2';
Dataset2006(21).leftImage='view1.png';
Dataset2006(21).rightImage='view5.png';
```

```
Dataset2006(21).groundTruthL2R='disp1.png';
Dataset2006(21).groundTruthR2L='disp5.png';
Dataset2006(21).dispMax=73;
Dataset2006(21).scale=3;
```

Bibliography

[1] R. Cooper. (1995) Magic Eye How to See 3D. [Online]. http://www.vision3d.com/stereo.html

[2] D. Scharstein and R. Szeliski, "A taxonomy and evaluation of dense two-frame stereo correspondence algorithms," *International Journal of Computer Vision*, vol. 47(1/2/3), pp. 7-42, Apr. 2002.

[3] P. F. Felzenszwalb and D. P. Huttenlocher, "Efficient Belief Propagation for Early Vision," *International Journal of Computer Vision*, vol. 70, no. 1, pp. 41-54, Oct. 2006.

[4] Y. Boykov, O. Veksler, and R. Zabih, "Fast Approximate Energy Minimization Via Graph Cuts," *IEEE Transaction on Pattern Analysis and Machine Intelligence*, vol. 23, no. 11, pp. 1222-1239, Nov. 2001.

[5] C. Murphy, et al., "Low-Cost Stereo Vision on an FPGA," in *IEEE FCCM*, April 2007, pp. 333-334.

[6] Y. Ruigang, M. Pollefeys, and L. Sifang, "Improved Real-Time Stereo on Commodity Graphics Hardware," in *IEEE CVPR Workshop*, June 2004, pp. 36-44.

[7] L. D. Stefano, M. Marchionni, and S. Mattoccia, "A PCbased real-time stereo vision system," *International Journal of Machine Graphics and Vision*, vol. 13, no. 3, pp. 197-220, Jan. 2004.

[8] G. v. d. Wal, M. Hansen, and M. Piacentino, "The Acadia vision processor," in *Proceedings of 5th International Workshop on Computer Architecture for Machine Perception*, Padova, Italy, 2001, p. 31–40.

[9] L. Wang, M. Liao, M. Gong, R. Yang, and D. Nister, "High-Quality Real-Time Stereo Using Adaptive Cost Aggregation and Dynamic Programming," in *3DPVT*, 2006, pp. 798-805.

[10] T. Kanade, A. Yoshida, K. Oda, H. Kano, and M. Tanaka, "A Stereo Machine for Video-Rate Dense Depth Mapping and Its New Applications," in *Proceedings of the 1996 Conference on Computer*

Vision and Pattern Recognition (CVPR '96), 1996, p. 196.

[11] J. Woodfill and B. V. Herzen, "Real-Time Stereo Vision on the PARTS Reconfigurable Computer," in *The 5th Annual IEEE Symposium on FPGAs for Custom Computing Machines*, Napa Valley, CA, USA, 1997, pp. 201-210.

[12] K. Konolige, "Small vision system. hardware and implementation," in *Proc. International Symposium on Robotics Research*, Hayama, Japan, 1997, pp. 111-116.

[13] J. I. Woodfill, G. Gordon, and R. Buck, "Tyzx DeepSea High Speed Stereo Vision System," in *Conference on Computer Vision and Pattern Recognition Workshop, 2004 (CVPRW '04)*, June 2004, p. 41.

[14] (2008, Sep.) STOC Stereo on a Chip. [Online]. http://www.videredesign.com/vision/stoc.htm

[15] Y. Jia, X. Zhang, M. Li, and L. An, "A Miniature Stereo Vision Machine (MSVM-III) for Dense Disparity Mapping," in *Proceedings of the Pattern Recognition, 17th International Conference on (ICPR'04) Volume 1 - Volume 01*, 2004, pp. 728-731.

[16] (2008, Sep.) Bumblebee®2 Stereo vision camera by PointGrey research. [Online]. http://www.ptgrey.com/products/bumblebee2/

[17] H. Hirschmuller and D. Scharstein, "Evaluation of Cost Functions for Stereo Matching," in *IEEE CVPR*, June 2007, pp. 1-8.

[18] H. Hirschmuller, "Improvements in Real-Time Correlation-Based Stereo Vision," in *Proceedings of the IEEE Workshop on Stereo and Multi-Baseline Vision (SMBV'01)*, Washington, DC, USA, 2001, p. 141.

[19] S. Ahuja, B. Khaleghi, and Q. M. J. Wu, "An improved real-time miniaturized embedded stereo vision system (MESVS-II)," in *IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops, 2008 (CVPRW'08)*, Anchorage, Alaska, 23-28 June 2008, pp. 1-8.

[20] S. Ahuja, B. Khaleghi, and Q. M. J. Wu, "A New Miniaturized Embedded Stereo-Vision System (MESVS-I)," in *Canadian Conference on Computer and Robot Vision, 2008 (CRV '08)*, 28-30 May

2008, pp. 26-33.

[21] (2008, Sep.) Selecting Processors for Video Applications. [Online]. http://www.bdti.com

[22] D. J. Katz and R. Gentile, *Embedded Media Processing*. New York, USA: Elsevier, 2006.

[23] (2008, Sep.) High-Speed Board Layout Guidelines. [Online]. http://www.altera.com/literature/an/an224.pdf

[24] N. Gianotti. (2008, Jun.) Fully automated PCB test: in-line automated loading and unloading allows low-volume, high-mix product testing to continue around the clock, Printed Circuit Design & Manufacture. [Online]. http://www.entrepreneur.com/tradejournals/article/165362399.html

[25] (2008, Sep.) Camera Calibration Toolbox for Matlab. [Online]. www.vision.caltech.edu/bouguetj/calib_doc/

[26] R. Zabih and W. John, "Non-parametric local transforms for computing visual correspondence," in *ECCV*, 2004, pp. 151-158.

[27] B. Cyganek, "Comparison of Nonparametric Transformations and Bit Vector Matching for Stereo Correlation," in *Combinatorial Image Analysis, Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2005, vol. Volume 3322/2005, pp. 534-547.

[28] J. Banks and P. Corke, "Quantitative Evaluation of Matching Methods and Validity Measures for Stereo Vision," *The International Journal of Robotics Research*, vol. 20, no. 7, pp. 512-532, Jul. 2001.

[29] T. Kanade, H. Kano, and S. Kimura, "Development of a video-rate stereo machine," in *Image UnderstandingWorkshop*, Monterey,CA, 1994, p. 549–557.

[30] L. D. Stefano, M. Marchionni, S. Mattoccia, and G. Neri, "A Fast Area-Based Stereo Matching Algorithm," *Image and Vision Computing, Proceedings from the 15th International Conference on Vision Interface*, vol. 22, no. 12, pp. 983-1005, Oct. 2004.

[31] L. D. Stefano, M. Marchionni, S. Mattoccia, and G. Neri, "A Fast Area-Based Stereo Matching Algorithm," in *6th IAPR/IEEE International Conference on Pattern Recognition (ICPR 2002)*, Quebec City, Canada, August 11-15, 2002.

[32] G. Egnal and R. P. Wildes, "Detecting binocular half-occlusions: empirical comparisons of five approaches," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 24, no. 8, pp. 1127-1133, Aug. 2002.

[33] B. Khaleghi, *Miniaturized Embedded Stereo Vision System (MESVS)*. Windsor, ON, Canada: MASc. Thesis, University of Windsor, 2008.

[34] D. Scharstein and R. Szeliski, "High-accuracy stereo depth maps using structured light," in *IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR 2003)*, vol. 1, Madison, WI, June 2003, pp. 195-202.

[35] D. Scharstein and C. Pal, "Learning Conditional Random Fields for Stereo," in *IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR 2007)*, Minneapolis, MN, June 2007, pp. 1-8.

[36] (September,) Estimating Power for ADSP-BF561 Blackfin® Processors, Analog devices Engineer-to-Engineer note 293. [Online]. www.analog.com

[37] M. J. Hannah, "Computer Matching of Areas in Stereo Images," in *PhD Thesis, Stanford University*, 1974.

Vita Auctoris

Siddhant Ahuja was born in 1984 in India. He went on to the University of Windsor, where he obtained a B.A.Sc. in Electrical and Computer Engineering in 2007. He is currently a candidate for the Master of Applied Science degree in Electrical and Computer Engineering at the University of Windsor, and plans to graduate in Spring 2009.