MEASURING CHARACTERISTICS OF RILL EROSION USING IMAGE PROCESSING TECHNIQUES

By

CHRISTINE THERESA RICE

Bachelor of Science in Agricultural Engineering

Oklahoma State University

Stillwater, Oklahoma

1985

Submitted to the Faculty of the Graduate College
of the Oklahoma State University
in partial fulfillment of the requirements
for the Degree of
MASTER OF SCIENCE
December, 1987

Thesis
1987
R495m
cop. 2

OKLAHOMA STATE UNIVERSITY LIBRARY

MEASURING CHARACTERISTICS OF RILL EROSION USING IMAGE PROCESSING TECHNIQUES

Thesis Approved:

Bruce Wilson
Thesis Adviser

Glenn Kranzler

Gregory J Hanson

Norman N. Durham
Dean of the Graduate College

1291044

PREFACE

This study is concerned with measuring soil profiles and velocity using image processing techniques. The main objectives are to develop the apparatus and algorithms and test the system. This research has been completed at Oklahoma State University in the Agricultural Engineering Department. The apparatus and testing materials were constructed in the Ag Engineering Laboratories. The profiles used in this research were preformed and the velocities were limited to known parameters. This project did not measure actual field properties, but extensions of this research will. The image processing techniques are limited to the capabilities of an IBM-AT complete with accessory boards.

I wish to extend my sincere thanks to my major advisor, Dr. Bruce Wilson, for his patience, guidance and assistance throughout the entire cource of study. Appreciation is also expressed to the other committee memberrs, Dr. Glenn A. Kranzler and Dr. Greg Hanson for their advice and suggestions and their assistance in the preperation of the final manuscript.

A special note of thanks is given to Mark Appleman who was responsible for writing the majority of the software for the system and being available when problems occurred. His assistance and talents have been much appreciated.

Thanks is also extended to Mr. Wayne Kiner and his staff, especially Robert Harrington, at the Agricultural Engineering Laboratory for their assistance in constructing the system apparatus.

I would also like to thank my husband Chris, for his understanding and patience thoughout this research, especially during the final stages.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

As water strikes the earth and flows over the land surface, soil particles are detached and transported. This process is generally referred to as water erosion. Although erosion is a natural geological process, society has greatly accelerated erosion rates with land disturbing activities associated with agriculture, urban development, silviculture and surface mining.

Water erosion is damaging in many ways. Soil depth is decreased; plant nutrients are removed; texture is changed; structure is degraded; productive capacity is reduced; and fields are dissected. Sediments produced by erosion pollute streams and lakes and are deposited on bottom lands and in channels and reservoirs.

The most apparent damage caused by water erosion is the removal of soil. Grant (1975) has indicated that erosion can vary from less than 1 metric ton/hectare/year from land covered with perennial vegetation, either grass or trees, to more than 450 metric tons/hectare/year from bare, cultivated fields. Although any soil loss is a concern, topsoil loss is most important (Troeh et al., 1980). Topsoil is generally more friable and more permeable to water, air and roots than deeper soil, and contains more organic matter and fertility than subsoils.

DeBoodt and Gabriels (1980) researched the severity of erosion on a world wide basis. Virtually every area in the world where food and fiber are produced must deal with water erosion. Of course the problem is worse in high rainfall areas, but not necessarily absent in low rainfall areas. In 1978, scientists from North

America, Africa, Australia and Europe met in Belgium to assess the world wide water erosion problem; all agreed that soil erosion is a universal threat to present and future crop production (DeBoodt and Gabriels, 1980).

Soil erosion and subsequent deposition are also nonpoint pollution sources. They adversely affect the quality of our fluvial systems by changing the aquatic life in streams and rivers, reducing the storage capacity of reservoirs and lakes, clogging navigable waterways, and transporting land-applied chemicals which otherwise would not enter the stream ecosystem. Erosion related pollutants have been estimated to impose net damages of $3.2 to $13 billion per year in the United States, with a single value estimate of $6.1 billion (Clark et al., 1985). Considering these impacts, it is not surprising that the USGS Water Resource Division Memorandum No. 85.80 places a highest priority in expanding the data base on the processes governing erosion, sediment transport and sediment deposition.

As the result of advancing technologies, innovative instrumentation techniques for gathering and expanding the erosion data base are possible. These techniques have the potential to gather data more accurately and efficiently than current methods allowing for more comprehensive analyses of erosion processes. An important advancing technology is low cost image processing systems. The application of this technology in quantifying erosion processes is the general thrust of this study.

The specific objectives of this study are:

1. To design and develop an apparatus to measure soil profiles and water velocity using image processing techniques,
2. To test the system for measuring soil profiles using rigid, well-defined objects of known size and shapes and using actual soil profiles,
3. To test the system for measuring water velocities using known velocities of small wooden beads.

CHAPTER II

LITERATURE REVIEW

Introduction

Advances in reducing water erosion are ultimately linked to obtaining a better understanding of the erosion process. Theoretical modeling can provide valuable insight into this process. Nevertheless, this work still relies on comparisons between predicted and measured values. As more fundamentally-based erosion models are proposed, the need to measure fundamental processes more accurately increases. This places a greater demand on instrumentation systems.

In this chapter, a brief overview of erosion research will first be given to provide a historic perspective and to underline important physical processes. The remaining sections will focus on current instrumentation techniques for measuring soil erosion processes. Emphasis will be on techniques for measuring soil profiles and for estimating flow velocities in rills.

Overview of Erosion Research

History

The first scientific investigations of erosion were carried out by the German soil scientist Wollny, between 1877 and 1895 (Hudson, 1981). Small plots were used to measure the effects of vegetation and surface mulches on rainfall interception and the deterioration of soil structure by erosion. Wollny also studied the effects of soil type and slope on erosion. In the United States, isolated cases of farmers implementing conservation practices were reported as early as 1850. Implementation of conservation practices gradually increased to the turn of the century. In 1907, the United States Department of Agriculture declared an official policy of land protection (Hudson, 1981). The United States is now a leader in soil erosion research.

Bennet (1939), with the help of funds from Congress, was able to establish a network of ten field experiment stations between 1928 and 1933 to study runoff and erosion. During the next decade, this program expanded until forty-four stations were operating. This program was primarily experimental. Early theoretical work was done by Ellison (1944) who analyzed the mechanical action of raindrops on soil.

In 1954, a national study was started to correlate the results of all the field experiments started by Bennett (Wischmeier, 1955). As a result of this study, the main features in the erosion process were identified and mathematically enumerated in the Universal Soil Loss Equation (Wischmeier et al., 1958). This work has had a major impact on the quantitative investigation of soil erosion.

Erosion Process

There are several types of water erosion including sheet erosion, rill erosion, gully erosion and streambank erosion. Sheet erosion is the removal of thin layers of soil by water acting over the entire surface area. Raindrop splash and surface flow are the mechanisms for detachment and transport of soil of sheet erosion. Raindrop splash is the primary detaching agent, and flow is the primary transporting agent (Troeh et al. 1980).

Schwab et al. (1966) defines rill erosion as the removal of soil by water from small but well-defined channels or streamlets. Rills are formed by the concentration of overland flow. Rill erosion is usually the form of erosion in which most of the soil erosion occurs. In contrast to sheet erosion, both detachment and transport of soil in rills are dominated by runoff characteristics.

When erosion channels become too large to be removed by ordinary tillage, they are then called gullies. Gullies are considered to be active as long as erosion keeps the sides bare of vegetation, and inactive when they have been stabilized by vegetation.

Sheet, rill and gully erosion are active only during or immediately after rainstorms. Erosion along the banks of perennial streams occurs both during and between rainstorms. Stream banks erode either by runoff flowing over the side of the stream bank or by scouring and undercutting below the water surface. Bottom-land soils damaged by streambank erosion are usually more productive than any other soils in the area (Troeh et al., 1980).

Measurement and Experimentation

Data on soil erosion and its controlling factors can be collected in the field or, for simulated conditions, in the laboratory (Morgan, 1986). Attempts have been made to distinguish between measurement and experimentation (DePloey and Gabriels, 1980). DePloey and Gabriels (1980) defined measurements as the steps used to determine erosion rates with observed data. They are not conducted to study erosion mechanics. According to DePloey and Gabriels (1980), experiments are conducted to obtain a better understanding of the erosion process itself. Since experiments generally also involve measurements, it is difficult in practice to separate the two.

Measurements are subject to error. No single measurement of soil loss can be considered as the absolutely correct value (Morgan, 1986). Errors are usually assessed in terms of variability. This requires replicating the experiment several times to determine the mean value of soil loss. In a review of field and laboratory studies of soil erosion, Beasley et al. (1984) found typical values of 13 to 40 per cent for the coefficient of variation for soil loss.

Experiments should be set up in such a way that they can be easily understood and repeated by other workers. Because of the natural variability in soils, it is sometimes difficult to have replicate runs. Additional errors may also arise due to different operators or to slightly different equipment, for example, different rainfall simulators (Morgan, 1986).

Techniques of Measuring Soil Profiles

Introduction

The soil surface profile is an important characteristic of erosion, especially rill erosion. In the last fifty years many rillmeter and profile meters have been developed, resulting in a variety of ways to measure soil profiles. The following sections will describe the three most common techniques used in the United States: pin displacement units, height tranducers (probes), and non-contact profile meters.

Pin Displacement Units

Pin displacement units measure soil profiles by the displacement distance of pins as they are moved from a reference level to the soil surface. Although the basic principles for pin displacement measurements are the same, automation and technology used to run the system, record and analyze the data vary with different studies. An example of a pin displacement unit can be seen in Figure 1.

Surface roughness for tillage effectiveness was measured by Kuipers (1957) using a board holding twenty vertical probes at 100 mm spacings. After these probes were lowered to the soil surface, twenty elevation readings were collected manually. The board was then moved to the next site at fixed intervals, depending

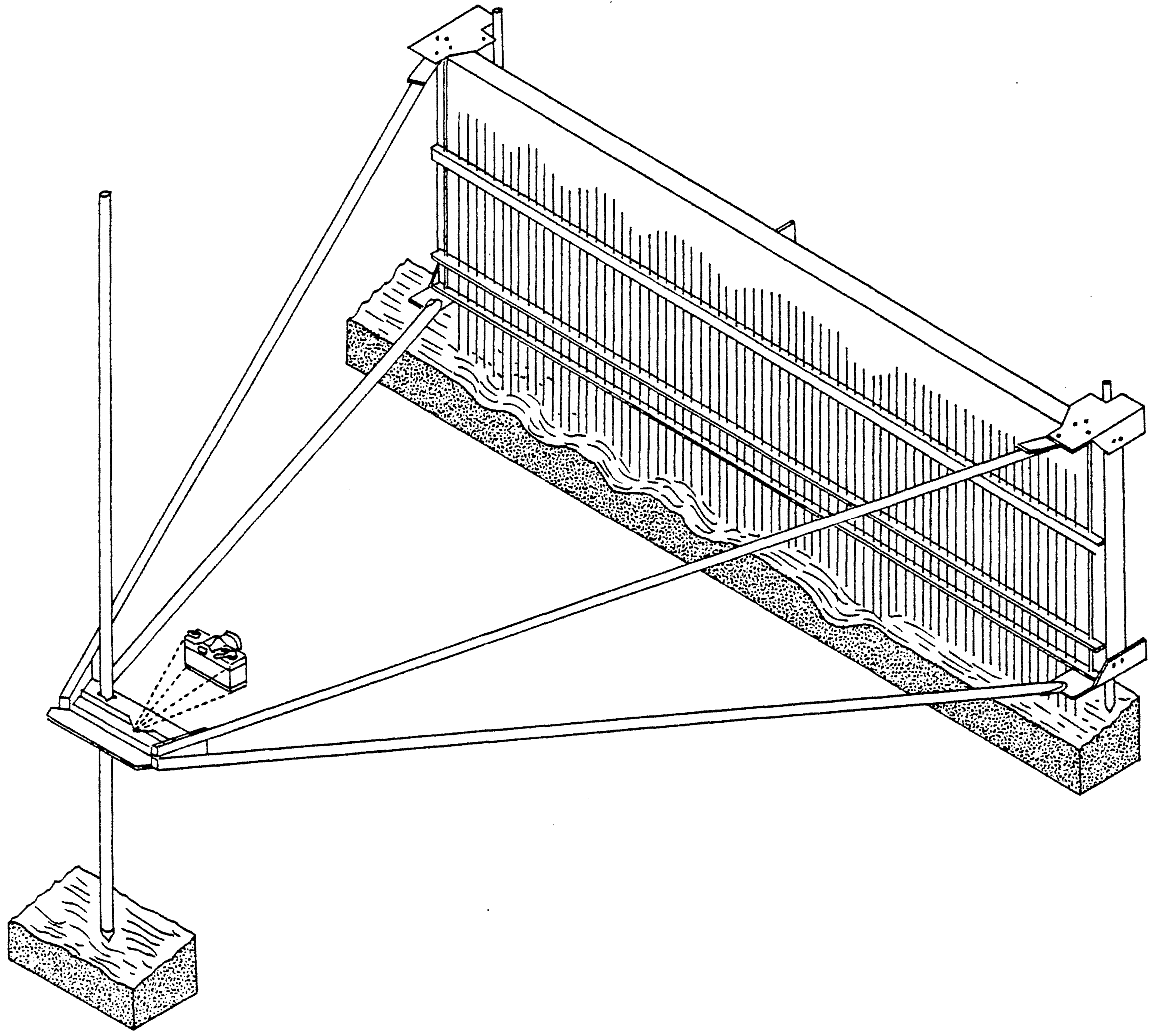


Figure 1. Pin Displacement Unit

on the surface area to be covered. Burwell et al. (1963) used a point quadrant instrument similar to that of Kuipers (1957) to measure soil surface elevation before and after preplant tillage. The soil surface elevations were measured to the nearest 0.25 cm on a 5.08 cm by 5.08 cm grid for a 1.02 m by 1.02 m area. Eighteen measuring pins were gently lowered until all pins were resting on the soil surface. The height measurements were read from a scale board at the top of the measuring pins. The measuring pins were then raised and advanced 5.08 cm and lowered again to obtain a three-dimensional representation of the surface.

Curtis and Cole (1972) measured soil loss from surface mined lands using a pin displacement unit. Their unit was a frame holding a row of 40 pins set 30 mm apart and arranged so they could move freely, vertically. The device was placed on preinstalled angle iron reference stakes. The pins were then lowered to the surface and a graph of a 1.2 m profile was exhibited. The graph was then recorded on film for later tabulation. In a similar manner, Foster and Meyer (1972) developed two rill meters which were used for estimation of soil movement in soil erosion studies. The profiles were determined by lowering the pins onto the soil and photographically recording the data. The pin spacings for the meters were 6.4 mm over 190 mm and 25 mm over 3.5 m.

McCool et al. (1976) reported on the use of a pin type instrument for measuring soil loss in rills. The instrument used 3 mm stainless steel pins on 12.5 mm centers. The working width of the unit was 1.83 m. The pin readings were recorded photographically and analyzed at a later time.

Moore and Larson (1979) measured the soil surface profile to estimate micro-relief surface storage. The plot size was 102 cm by 102 cm. Spot measurements of surface elevation were taken on a 5 cm by 5 cm grid using the surface relief meter described by Allmaras et al. (1966). A horizontal bar holding 18 pins was lowered manually by a crank device until each pin touched the surface. Individual cross sections were recorded photographically and digitized manually.

McCool et al. (1981) revised and updated his rillmeter to make rill erosion measurements on cultivated fields and runoff plots. The pin spacing was the same as in his previous study, but the measuring rods were constructed of aluminum alloy welding rods, and the frame was stronger and more maneuverable. Photographic data were processed using a manual electronic digitizer which was connected to a desk-top calculator. A calculator program was written to determine the amount of soil eroded using the trapezoidal rule.

A microprocessor automated rillmeter was developed by Radke et al. (1981). The rillmeter measured 312 surface elevations with sensing rods arranged in three rows of 104 rods each. The rods were spaced 1 cm apart within the row and 5 cm apart between rows. An electric motor was used to lower the grid of sensing rods. Electrical contacts were made when the rods were touching the soil surface. A micro-processor then scanned and stored the platform position of each sensing rod. Data were stored on magnetic casette tape.

Highly-automated pin type rillmeters were developed by Hirschi et al. (1984) to measure soil surface heights. Two similar rillmeters were constructed of different sizes and pin spacings. One meter had a 13 mm pin spacing over a 1 meter width (72 pins), and the other had a 64 mm spacing over a 4.5 meter width (70 pins). Each had a measurement accuracy of $\pm$ 1mm. The mechanical movement of the pins was the same as the meter utilized by Moore (1979). The pins were lowered using a remote control circuit, thus eliminating the necessity of walking on the plot. Electronic sensing was used to determine the pin location. A stainless steel contact on the top of each pin connected wires in a parallel network to a voltmeter. The voltmeter was used to sense the voltage drop along a parallel network wire between the pin location and the lower bar holding the pins. Computers were used to control the rillmeter, take data, process the data and stop the data collection.

Height Transducers

An alternative method to pin displacement units is to use a single height transducer mounted on a carriage for lateral movement which moves across a plot. This method is typically similar to a pin displacement unit, in that a probe is lowered to the soil surface from a reference height. It differs, however, because only a single probe is used, and because contact between the probe and soil is the switching mechanism that determines the height readings. Figure 2 shows the basic concept of a height transducer.

An automated soil surface profile-meter was developed by Schafer and Lovely (1967). This system would automatically make and record a large number of point elevation readings over a distance of 2.1 m at 25 mm intervals. A prodding device rolled laterally along a horizontal beam extending across the frame length. Height readings were obtained by lowering the prod until it contacted the soil surface. A sensor would then actuate the prod bottom when the soil was touched. A recorder would record the distance the prod travelled to reach the surface. This same concept was extended by Currence and Lovely (1970) into a fully automatic recording profile-meter. This profile-meter allowed height readings to be taken on a 25 mm grid over an area of 1.5 m by 2.0 m. The travel distance of the probe was recorded using a card punch. Height readings recorded by the profilometer were accurate to $\pm$ 0.127 cm.

Mitchell and Jones (1973) used a device similar to Currence and Lovely (1970) to measure soil surface profiles. The device consisted of a carriage probe unit, a power supply control and a recording unit. It was designed to measure a 2.54 cm by 2.54 cm grid for a surface of 0.91 square meters. The movement of the probe over the test area could be either completely automated or manually controlled. At each measurement point, the probe was driven downward until the sensing rod touched

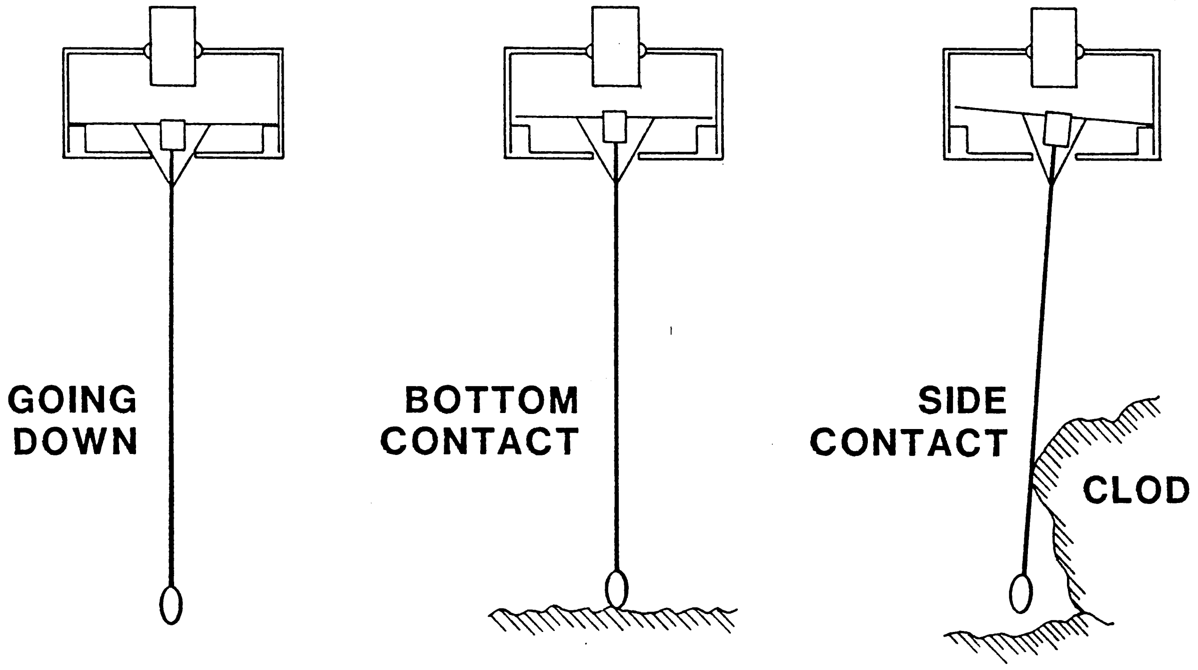


Figure 2. Height Transducer Probe

the surface. When this happened, a snap action switch in the probe was activated by the pressure of the movable sensing rod on the soil surface. At that instant, the probe returned to the probe carriage and moved to the next position. The measuring system within the probe carriage was a ten turn rotational potentiometer which would send a voltage signal to the recording system when the probe was actuated. The recording system would then hold the voltage signal, convert it to a digital signal and place the digital signal on paper tape.

Semi-automatic micro-relief meters to relate surface roughness to hydraulic roughness have been investigated by Heermann et al. (1969) and by Merva et al. (1970). To study shallow overland flow, profiles were recorded at close intervals. Heermann et al. (1969) obtained roughness measurements longitudinally down a furrow bottom at intervals of 3.2 mm for a distance of 2.9 m. Each digit on the vertical potentiometer was equivalent to 0.001 cm. The mechanical cycle was automatic and continuous. Merva et al. (1970) illustrated the difference between macro- and micro-surfaces. Spectral density analysis was applied to micro-surface profiles measured at 20 mm spacings over a distance of 3.0 meters in grassland. Anisotropy of surfaces was estimated by comparing the spectra of cross and down slope profiles.

Henry et al. (1980) developed a device for measuring soil surface profiles electromechanically. Using electronic controls and a battery powered printer, elevations and their horizontal locations were automatically printed on paper tape. The device consisted of a frame which served as a track for a horizontally powered carriage. Mounted on the carriage was a soil sensing probe which was driven up and down with a low inertia motor. A contact circuit controlled the probe motor to drive the probe down until either the bottom or the side of the sensor wire touched the soil. The motor would then reverse its direction. As soon as the sensor wire cleared the soil, the motor would again go in the downward direction. This process caused the probe tip to follow the contour of the soil surface. The output was sent to a battery powered strip chart recorder or a digital voltmeter and DC printer.

Linear profiles of stabilized sand surfaces were measured using a linear variable differential transformer (LVDT), which was designed by Podmore and Huggins (1981). Elevation measurments were required to relate physical roughness with hydraulic roughness. A step size of 0.25 mm was chosen so that the effects of coarse sand and larger roughness elements could be measured. The profilemeter consisted of a frame and a cross carriage which held the vertical probe. The cross carriage would allow a total horizontal distance of 1.85 meters to be measured. The LVDT consisted of a central core and various coils sending a transformed output in current. The signal was then converted to a voltage output which resulted in a linear relationship between core position and output voltage. The LVDT core was bonded to a sensing probe and measured sharply varying surface profiles with resolution of $\pm$ 0.005 mm. Data were collected on magnetic tape and analyzed using a large central computer.

Non-Contact Meters

Another type of soil profile meters is one in which the measuring device never comes in contact with the soil surface. This relatively new approach has become more economically feasible in recent years because of the reduction in cost of hardware equipment.

Harral and Cove (1982) developed an optical displacement transducer for the measurement of soil surface profiles. The transducer has a fast response opto-electronic displacement moniter with a working range of $\pm$ 150 mm for a point 600 mm from the sensing head. The device collected light from an illuminated spot using a semiconductor laser diode on the target surface and focused this onto a position-sensing photo-diode, giving an output related to the position of the target. As the surface level changed, the focused image would move on the detector surface. The resolution of the system was $\pm$ 6 mm.

An automated non-contact micro-relief meter was developed by Romkens et al. (1982) to measure profile elevations of the soil surface in field situations. The meter consisted of an optical probe which scanned the soil surface at a known tracking height in predetermined transects. Ball screws made horizontal movement of the carriage and vertical movement of the probe possible. A servo motor moved the probe vertically, while a 12 V DC motor moved the carriage. Displacements were recorded by encoders, which relayed electrical pulses for directional movement via digital subtractors to a cassette tape data logger. The plot area in which the probe system was able to move was 1 m by 1.15 m. The electronic components of the profiler consisted of a super pulser, an infared LED with phototransistor, a servo controller, a pulse counting system and a recording system. The super pulser generated an analog voltage output which became more negative as the probe approached the soil surface. The probe would move up and down depending on the various voltage readings. The probe would stay at a constant height above the soil surface. The necessary location adjustment of the sensor to maintain that height was monitored and recorded automatically. A 250 point per meter transect could be completed in about 4 minutes.

Another non-contact optical device was developd by Khorashahi et al. (1984) to measure soil surface elevations before and after artificial rainfall for erosion studies. The profiler consisted of a digital camera, a laser, and a mechanical drive train for horizontal movement. Calibration was done on the system using two artificial surface heights and the position of a camera-laser plate. Color and positions did not affect the distance measurements. The accuracy of the height measurements was within $\pm$ 1.95 mm. The surface that the system covered was 1.5 m by 1.5 m. The digital camera and laser were mounted on an adjustable stand. A stepper motor controlled by a logic board in a microcomputer was responsible for the horizontal movement. A prefabricated control board was used to interface the camera with the microcomputer. One section of this board controlled camera operation and received the data, and the other section manipulated and transferred the data to the microcomputer.

Techniques of Measuring Velocity in Rills

Introduction

Rills are usually very irregular in their cross section and grade, resulting in high spatial variabilities in hydraulic variables. Intense local velocities may result in significant erosion. Information on rill velocities is therefore needed to improve our understanding of the erosion process (Foster et al., 1984). Although there are many ways to measure fluid velocities, only those techniques that are readily applicable to flows in small open channels will be discussed. The techniques discussed are pitot tube measurements, hot film anemometry and dye measurements.

Pitot Tube Measurments

A pitot tube is a submerged tube which is oriented so that the axis is parallel to the flow of the fluid (Shames, 1982). The ambient pressure is measured through holes in the side of the tube, and the stagnation pressure which represents the total head is measured through a hole in the tip of the tube.

The pitot tube can be used to determine velocity using the following special form of the Bernoulli equation

$$v = \sqrt{\frac{2\Delta p}{\rho}}$$

in which v is the velocity, ρ is the density of the fluid and Δp is the difference in pressure between the tip of the tube and the side holes. In this way, the magnitude of velocity is determined in a simple and straightforward fashion (Albertson et al., 1960).

A major problem in the use of an ordinary pitot tube is to obtain proper alignment of the tube with flow direction. The angle formed between the probe axis and the flow streamline at the pressure opening should be zero, but many times the angle may not be constant. The flow may be fixed in either magnitude or direction. Beckwith et al. (1982) show that the pitot tube is particularly sensitive to yaw. Although sensitivity is influenced by orientation of both impact and static openings, the latter probably has the greater effect.

Vanoni (1946) found it necessary to measure water velocity when researching the transportation of suspended sediment by water. A pitot static tube was used with a diameter of 4.67 mm. The differential pressure on the tube was read to an accuracy of 0.03 cm on a water manometer.

Vanoni and Brooks (1957) also used pitot tubes to study roughness and suspended load of alluvial streams and to measure vertical velocity profiles. Pitot tubes of diameter 4.76 mm and 6.35 mm were used in their studies.

Hot Film Anemometry

In hot film anemometery, velocities are determined by changes in the electrical resistance of a thin film of carefully constructed material (i.e. platinum, tungsten) surrounding a cylindrical element. The film is heated above the ambient temperature of the surrounding fluid by passing an electrical current through a resistance material. Flow of the fluid over the hot film cools it by forced convection. The cooling of the film is a function of the velocity of the flow; temperature, density, viscosity, and thermal conductivity of the fluid; temperature, diameter, and length of the film backing material; and thickness of the film. If all but fluid velocity are kept constant, the heated film is a transducer for measuring velocities (Richardson and McQuivey, 1968).

The hot film anemometer has a very short response time, permitting it to pick up rapid fluctuations in velocity. Also, the probe of the device is very small, so rather than getting average values of velocity over a comparatively large region as in the case of the pitot tube, an average over a much smaller region can be taken and for all practical purposes, the measurements are considered valid for a point in the flow (Shames, 1982).

Richardson and McQuivey (1968) used a hot film anemometer to measure turbulence in water. The probes used in this study had a thin coating of quartz fused over the platinum to insulate the conductor from the fluid. The coating eliminated stability problems caused by electrolysis and conductivity through the fluid medium. A method was developed for measuring turbulence in extremely dirty water. The method was based on a hypothesis that dirt and air bubbles accumulating on the sensor decrease the mean voltage for a given velocity, but in the domain of frequencies encountered in water, do not affect the frequency response of the sensor to velocity fluctuations. The hypothesis was experimentally verified using hot film anemometers by comparing turbulence measurements made in clean and very dirty water.

Hot film anemometry and random signal analysis were used by Barfield et al. (1969) to measure the turbulent diffusivity in shallow open channel flows as affected by rainfall. Each quartz coated hot film probe required a unique calibration curve. Calibration was done in a flume using pitot tube velocities and observed voltages from the hot film probes. The hot film anemometers were calibrated to an 0.0076 m/sec.

Barfield and Henson (1971) discussed different calibration methods for hot film probes. Since calibration of the probes is one of the major problems associated with the use of the anemometer, this was an important topic to discuss. Several calibration methods were discussed including two methods developed by the authors.

In a laboratory study of rill hydraulics, Foster et al. (1984) used hot film anemometry to obtain velocity measurements and relationships. In addition, average velocity at a section was computed by dividing discharge rate at the section by flow area determined from water surface and rill cross-section elevations. A comparison of hot film velocities measured at several points and those obtained by average section velocities were within four percent.

Wilson and Barfield (1986) used a constant temperature anemometer unit to measure the turbulent characteristics of pond flows. The probe used was cylindrical and relatively insensitive to the direction of the fluid velocity which was a necessity due to the possible fluctuations in the pond's recirculation pattern. Analog data from the anemometer were converted to a digital form and stored in data files using an IBM PC computer. The data were analyzed to obtain estimates of mean velocities, mean square values of velocity fluctuation, Eularian time scales and turbulent diffusion coefficients.

Dye Method

Another method commonly used to measure water velocities is a technique involving dye. Hydraulic variables in streams and rivers have been widely identified using this method. Fluorescent dyes, utilized in dye dilution procedures, are economical, easy to handle and can be measured quantitatively in very low concentrations. However, characterization of hydraulic parameters using fluorometric techniques has received only limited use in upland areas.

While examining the effect of soil-surface configuration resulting from tillage tool marks on erosion, Young and Mutchler (1969) measured the flow velocity of water in small triangular furrows. The velocity was measured by injecting dye in the stream at certain points and timing the advance of the dye front. Mean flow velocity was also measured by Gilley et al. (1986) using a fluorometer. A slug of dye was injected into the channel and the amount of time required for the concentration peak to pass a downstream point was determined. A time-concentration curve resulted from continuous pumping of the sample through the fluorometer flow cell. Mean flow velocity was obtained by dividing travel distance by time of travel.

Line and Meyer (1978) measured average flow velocities along row furrows under intense simulated rainfall. An estimate of the velocity was obtained by introducing several drops of fluorescent dye onto the center of the flow surface and recording the time required for its peak to travel from a point 2 meters down to a point 8 meters down the furrow. Since dye-travel times determined by observation are subjective, a related laboratory study was conducted to correlate the field measured dye velocities to average flow velocities. Regression equations were developed and used to convert field dye results to average velocities.

CHAPTER III

INSTRUMENTATION SYSTEM

Introduction

A system has been developed to measure soil profiles and velocities in rills using image processing techniques. This system is incorporated into a large-scale laboratory apparatus designed for experiments that (1) require instrumentation techniques that are difficult to use in the field, (2) need control of erosion parameters to examine fundamental processes more accurately and/or (3) are conducted more efficiently in a laboratory setting because of cost and time constraints. The focus of this chapter is on the mechanical, electrical and structural components of the measurement system including the erosion table, soil profile measuring equipment and velocity measuring equipment. Details of the system's software are given in the next chapter.

Erosion Table

The erosion table is located at the edge of the Oklahoma State University Campus in one of the Agricultural Engineering Department's research shops. It has been designed to conduct erosion studies on a 2.4 m by 9.8 m surface. A view of the erosion table from the upslope end of the plot is shown in Figure 3. A side view of the erosion table is shown in Figure 4. The sidewalls are constructed of 2.9 cm plywood supported by a metal frame with appropriate bracing at approximately 1.2 m intervals. The plywood has been coated with a fiberglass sealant.

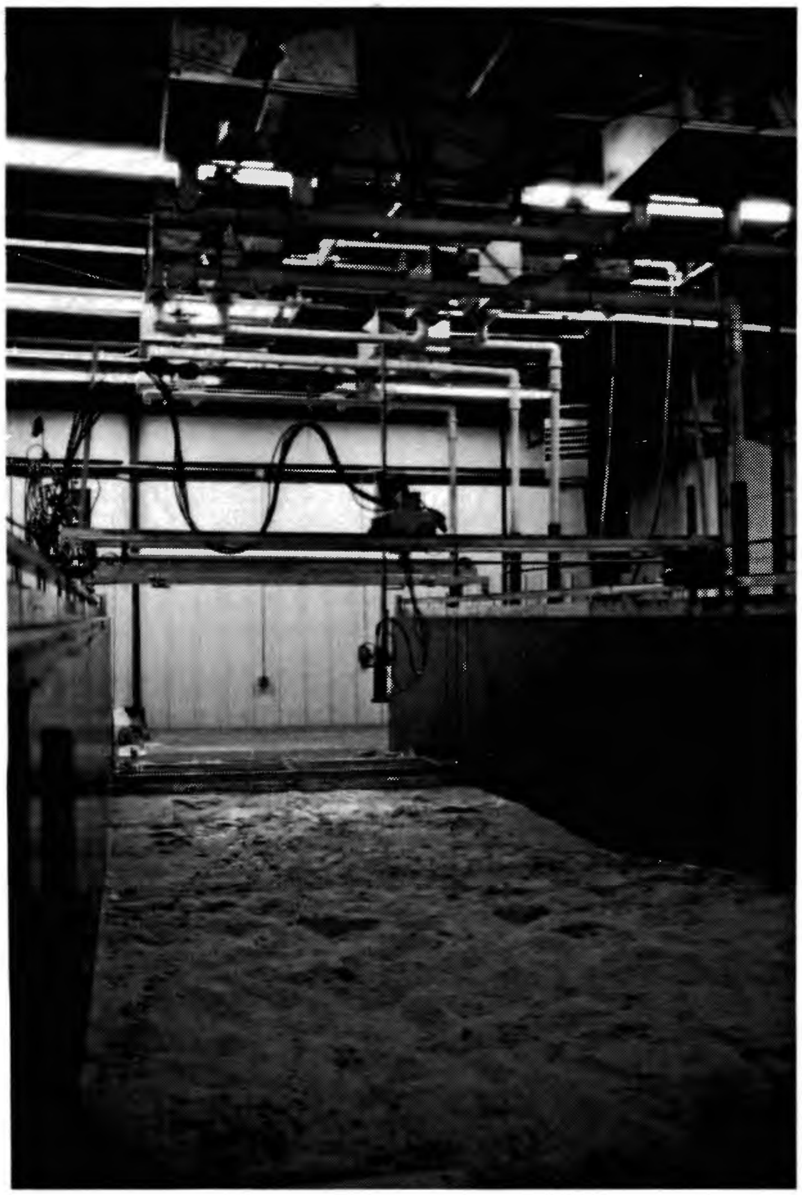


Figure 3. View of Erosion Apparatus

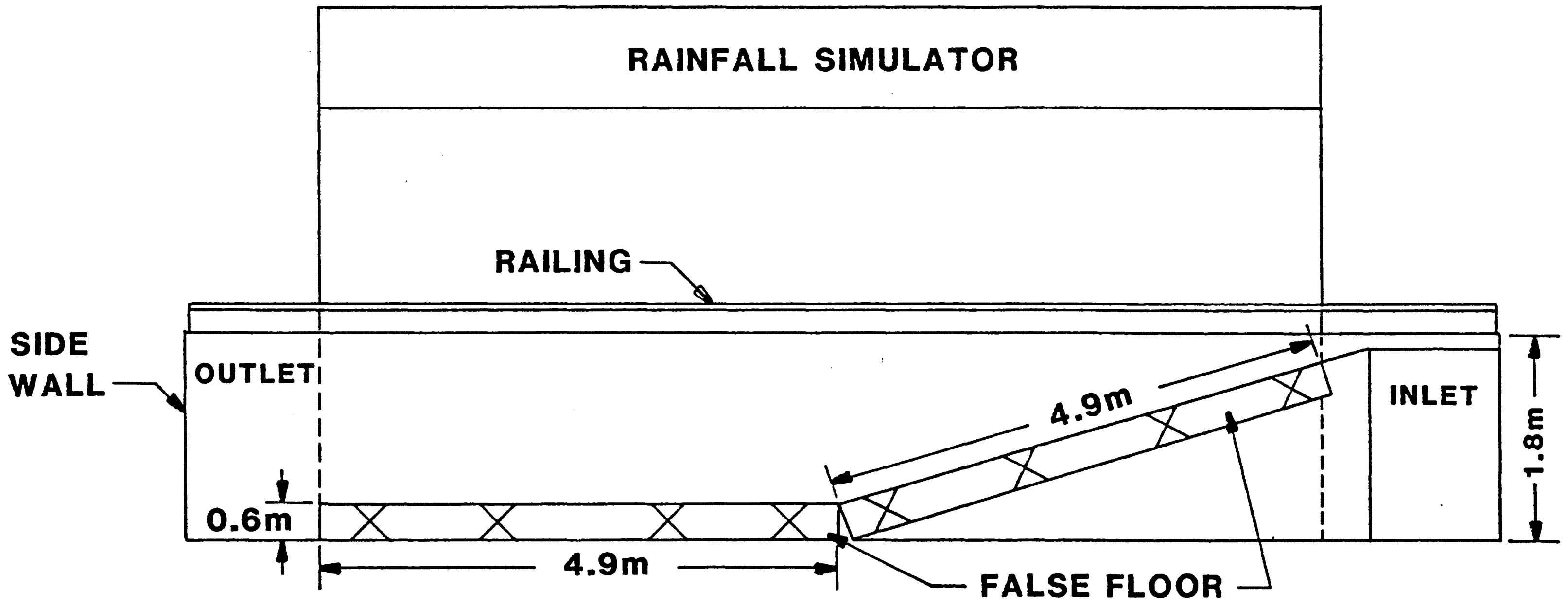


Figure 4. Side View of Erosion Table

As shown in Figure 4, two false floors support the erosion surface. The upslope floor is hinged so that it can rotate to obtain different slopes and spatially varying sideslopes. Fine tuning of surface profiles can be obtained by varying the depth of soil. The maximum possible uniform slope over the entire plot length is roughly 11 percent and roughly 19 percent for the adjustable section of the plot.

Located above the erosion table is a rainfall simulator. This simulator has been designed to duplicate natural rainfall by matching of kinetic energy and momentum factors. Further information on the erosion table and rainfall simulator is given by Wilson and Rice (1987).

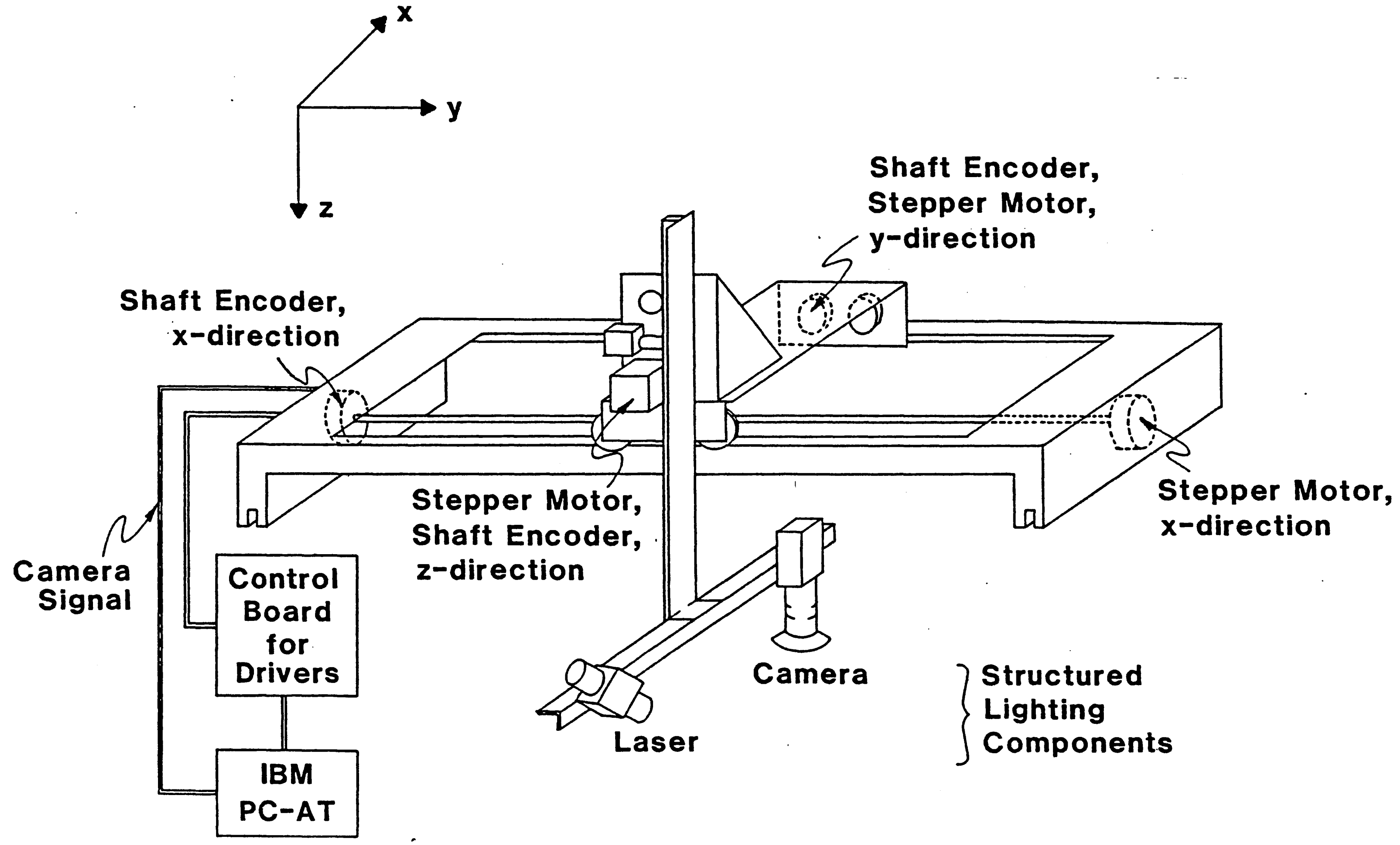
The focus of this research is to measure erosion processes occurring on the erosion table. An important component of this system is the instrumentation rack shown mounted over the surface in Figure 3. Details of this component and the image processing hardware are given in the next section.

Soil Profile Measuring Equipment

An important part of this study is the development of techniques for measuring surface topography. These measurements are made using a structured lighting technique. This technique requires that a well-defined stripe of light be projected onto the surface. The location of the stripe within the field of view of a camera is compared to some base value to obtain an elevation measurement. The support and movement of the light source and camera are done using the instrumentation rack. Image processing boards and software are used to analyze the data. The software component of this approach is described in the next chapter. Mechanical and hardware components are described here.

Mechanical Driving System

A schematic of the instrumentation rack is shown in Figure 5. The rack width


x
y
z
Shaft Encoder, Stepper Motor, y-direction
Shaft Encoder, x-direction
Stepper Motor, Shaft Encoder, z-direction
Stepper Motor, x-direction
Camera Signal
Control Board for Drivers
IBM PC-AT
Camera
Laser
Structured Lighting Components

is roughly equal to the width of the erosion plot (i.e., 2.4 m). On top of the rack is a platform that supports the mechanical components of the structured lighting equipment. This platform can move laterally across the plot and is used to move the structured lighting equipment vertically. Movement in the x, y and z directions is powered by three Cyber Research's ESH 088 stepper motors. These motors provide a torque of about 780 mN-m with 1.8 degrees per step. The stepper motors are driven by Cyber Research's 3-amp stepping motor driver boards (ESH 082) which in turn are operated using an IBM PC-AT with a Cyber Research's controller board (ESH 080).

Motion in the y and z directions is driven by precision racks and pinions. Figure 6 shows the driving mechanism for the z-direction. The racks (Reliance Gear Company R10A standard rack) have a maximum adjacent tooth-to-tooth error of 0.010 mm. The y-direction pinion has an outside diameter of 35.58 mm and travels 101.6 mm (4 inches) per revolution. One step of the stepper motor therefore results in a movement of 0.508 mm (0.02 inch). In the z-direction, a series of gears is used to allow for finer distance steps and to increase the holding and driving torque of the motor. In comparison to the y-direction, the gear reduction is 4:1, so that one step results in a vertical movement of 0.127 mm (0.005 inch). The maximum travel distance in the z-direction is roughly 1.2 m (4 ft). A solenoid switch is used as a brake in the z-direction to hold the structured lighting equipment in place when the power is off.

The instrumentation carriage must be capable of moving the entire length of the plot (i.e., 9.8 m). To reduce cost, movement in the x-direction is by a chain driven system. The carriage is mounted on adjustable aluminum tees that have been carefully leveled. A standard AMSI #25-lR roller chain is used to move the rack precisely and automatically. A gear is placed on the stepper motor and a shaft running down the center of the carriage is rotated to move the carriage. Chains are

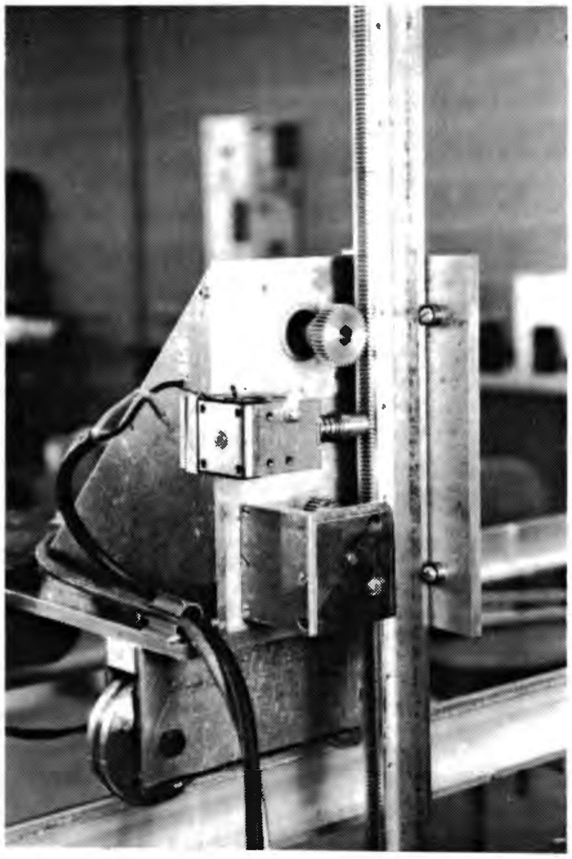


Figure 6. Vertical Direction Driving Mechanism

mounted to both sides of the erosion table and matching gears on each side of the carriage are used to keep both ends moving together. One step in the x-direction results in a movement of 0.38 mm (0.015 inch).

Travel distance in the x, y and z directions is also monitored using three Disc's Model 701FR-200-OCN-SS shaft encoders. These shaft encoders have a resolution of 200 pulses per revolution or 1.8 degrees of angular rotation. The shaft encoders are connected to the IBM PC-AT computer using the stepper motor driver boards previously discussed. By using appropriate software, stepper motors, and shaft encoders, the structured lighting equipment can be moved precisely to a desired (x,y,z) point over the plot.

Image Processing System

The main components of the structured lighting system are (1) a laser light to provide a well-defined stripe of light, (2) a video camera to sense the reflected light, and (3) computer boards in an IBM PC-AT to digitize and manipulate the image. These components are shown in Figure 5. The arrangement of the laser light and camera is shown in Figure 7. The horizontal distance between the laser and the camera is 0.43 m, and the laser is tilted at about a 45 degree angle.

The laser light is Newport Corporation's SLD-1008 diode laser line projector. It is a self-contained unit that has a laser light source and lens to produce a well-defined stripe of light in the near infared range of 770 to 820 nm. On special request, the laser has an optimum focus length of 0.6 m (2 ft) which corresponds to a striped line of 0.3 m (1 ft) length with a width less than 1 mm. The laser light has an intensity of 2 mW and runs on 12 volts DC.

A Hitachi KP 130 solid-state, black-and-white video camera is used in the instrumentation system. This camera has a sensor array with 384 horizontal and 485 vertical picture elements. The camera has an interlaced scanning system and a

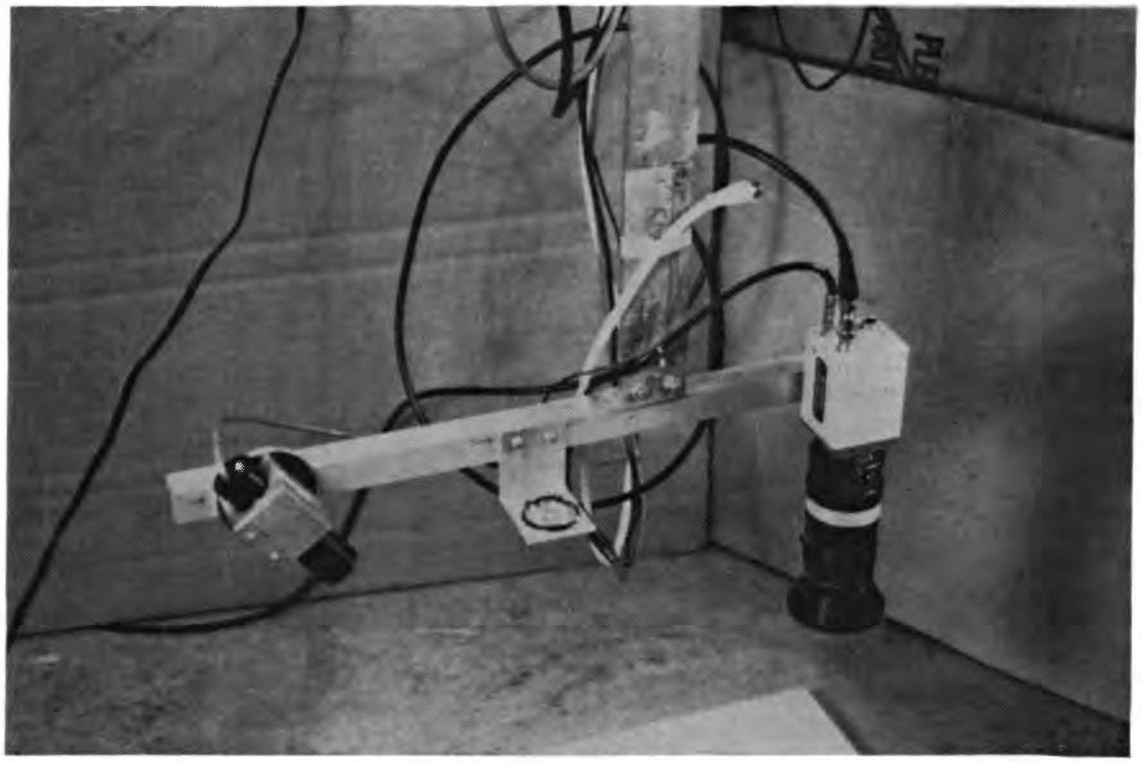


Figure 7. Structured Lighting Equipment

standard RS-170 output signal. It has the desired features of relatively low power requirements of 12 volts DC, low weight (1 lb) and small size (2.2" x 2.1" x 3.3").

Output signals from the camera are sent to a Data Translation DT-2851 frame grabber board located in an IBM PC-AT. This board has flash A/D converters that can digitize a video frame in 1/30 of a second. Its resolution is 512 lines by 512 pixels with 256 possible gray values. Limited processing on the board can be done such as frame averaging, frame addition and subtraction, and windowing. It also has two memory-map 256 Kbyte frame-store memory buffers to store two frames. An AST Advantage board with 1.5 Mbyte of RAM has been installed in the IBM PC-AT to store additional frames. The DT-2851 frame grabber can take input directly from a video camera or from a recorded tape using a video playback unit. Output from the frame grabber board is viewed on a Hitachi 12" black-and-white, solid-state VM-129 monitor.

The computational speed of the system is increased using a Data Translation DT-2858 auxiliary frame processor board. This board has high speed direct interface to the frame grabber memory, pipelined arithmetic performance of 2.5 million multiplications per second 700,000 divisions per second, and 2.5 million addition per second and supports NxM convolutions, frame averaging, normalization and histograming operations. The DT-2858 board is located in a slot next to the frame grabber in the IBM PC-AT.

The distance from the instrumentation rack to the soil surface can change significantly as the rack is moved, especially for steeply sloped surfaces. To keep the camera and laser light in focus, an ultrasonic distance measuring device has been placed on the bar holding the camera and laser as shown in Figure 7. The ultrasonic distance measurement system includes a sensor, a data acquistion card and software (purchased from ICS computer products). This equipment is used to obtain an average measurement of height in the vicinity of the camera. Based on this measurement, the structured lighting bar is raised or lowered using the vertical stepper motor to maintain a height of roughly 0.43 m above the soil surface.

Velocity Measuring Equipment

An important erosion parameter is the flow velocity in rills. Image processing techniques to measure this parameter are developed in this study. Velocity is determined by the movement of small wooden beads as they float past a camera. Two frames containing the beads are taken at different points in time. Velocity is then calculated from a measured displacement distance and the time interval between the frames. Software algorithms for this procedure are discussed in Chapter IV.

The camera, computer and image processing boards for the velocity measurements are the same as those used to measure soil profiles. Contrast between beads and background is enhanced by painting the beads with flourescent paint and using an ultra-violet bulb for a light source. The computational speed of the velocity measurement algorithm is too slow to run in real time. Therefore, the movement of the beads is first taped using an NEC high quality digital video cassette recorder (VCR). Once the process has been recorded, image processing algorithms are used to determine velocity.

Several sizes and types of beads were tested. Styrofoam beads were too light and had a tendency to attract to each other. Bead diameters any larger than 5 mm would not travel at the water velocity, frequently catching themselves on the bottom or sides of the channel. Wooden beads of approximately 5 mm diameter floated very well and were easy to handle. Different colors of flourescent paints were tried to obtain the greatest contrast using an ultra-voilet light source. Flourescent yellow paint was selected.

CHAPTER IV

IMAGE PROCESSING SOFTWARE

Introduction

The successful application of an image processing system depends upon its software. Various algorithms and routines must be written to manipulate and analyze data obtained with the hardware described in Chapter III. Different programs are used here for the soil profile and the velocity measurements. Both programs have been developed utilizing software functions supplied by Data Translation. This software support package (DT-IRIS) will be briefly discussed. Algorithms used for profile and velocity measurements will then be described. The testing of the measurement algorithms will be discussed in Chapter V.

DT-IRIS Software Functions

DT-IRIS is a comprehensive image processing support package for the DT2851 frame grabber board and the DT-2858 auxiliary processor board. The software package is composed of two sections: 1) a command driven tutorial program which provides an interactive image processing environment and 2) an imaging subroutine library package which provides a set of imaging functions callable from most popular high level languages. For this particular image processing system, the C language has been chosen because of its computational speed and popularity.

The DT-IRIS package includes important image processing techniques to manipulate images. Callable functions frequently used include routines for aquiring and displaying images, selecting frame buffers, modifying and selecting input and output lookup tables, and arithmetic operations such as adding and subtracting frames, and convolutions. Other important routines used extensively are those which do windowing and region operations, histograms and graphic overlays. DT-IRIS routines are accessible at link time.

Soil Profile Software Algorithm

Structured Lighting Concepts

General. Surface topography is measured in this research using structured lighting techniques. This approach is commonly used in industrial settings when depth readings need to be incorporated into machine vision systems (Jalkio et al., 1985; Swientek, 1986). The hardware components of our system are a laser source to project a well-defined stripe of light onto the surface, a camera to sense the reflected light, and image processing boards to digitize and manipulate images. Details of these components were previously given in Chapter III.

A three-dimensional schematic of a structured lighting system is shown in Figure 8. A laser source at (x_s,y_s,z_s) projects a well-defined stripe of light across a block of fixed height situated on a flat surface. The light is gathered through a lens located at (0,0,0) and focused on a sensor located at a vertical height of F behind the lens. The sensor image of the striped light is shown in the inset of Figure 8. The image is digitized into a square grid (512 x 512) of discrete picture elements or pixels. Since the light is striking the block at an angle, there is a difference in pixel locations between the top of the block and that of the flat surface. Differences in these locations can be used to determine the height of the block.

Geometry Considerations. Relating the difference in pixel locations to a block height is a geometry problem. To help clarify the geometry, two-dimensional views

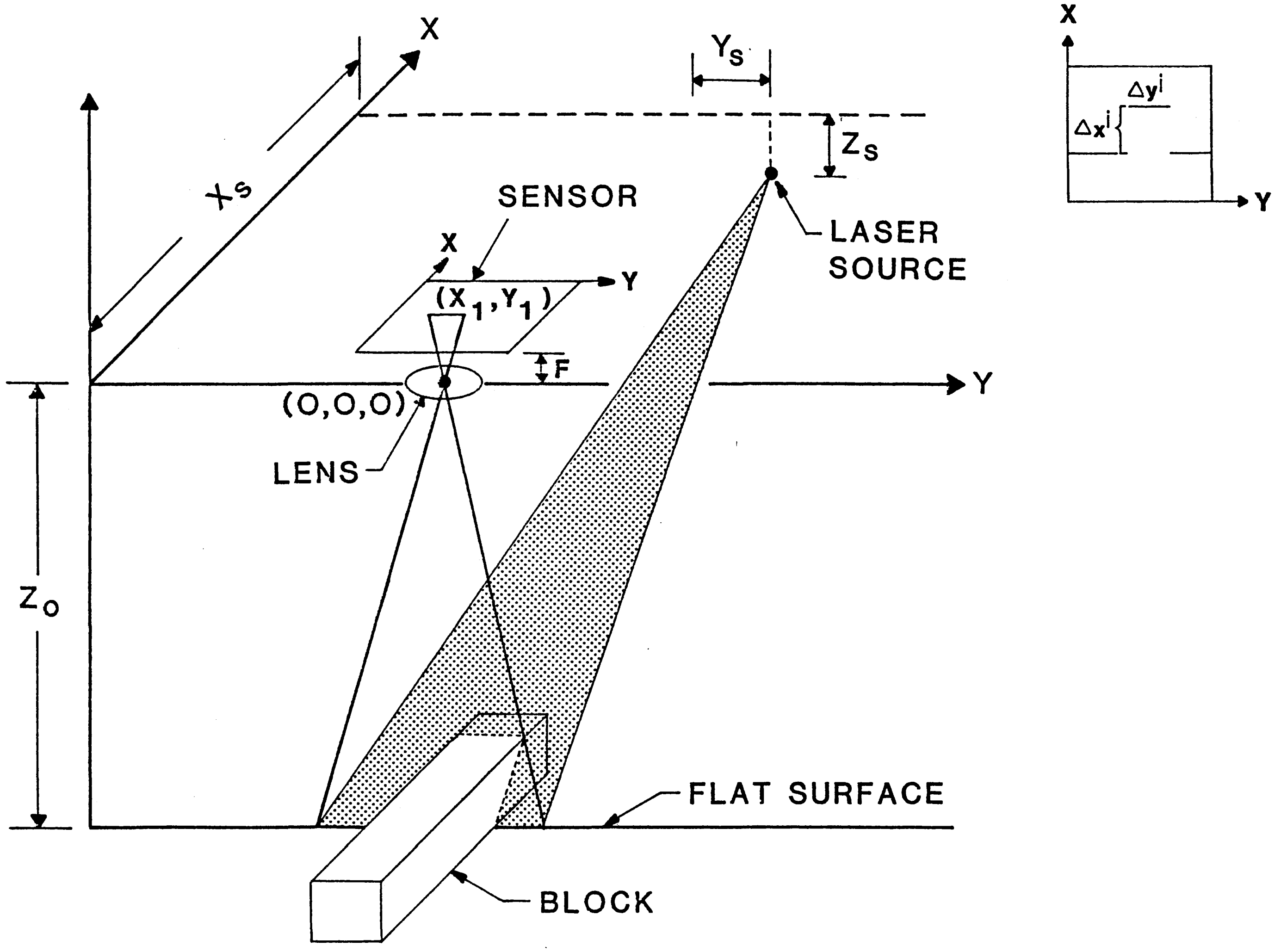


Figure 8. Schematic Illustrating Structured Lighting Concepts

of the y-z plane and the x-z plane are shown in Figure 9a and Figure 9b, respectively. From Figure 9a, the pixel location for a point (y_o, z_o) on the surface can be related to the location of the light source (assumed to be a point), the angle of the source and the vertical height. The angle of the source is defined as

$$\tan\theta_y = \frac{y_s - y_o}{z_o - z_s} \tag{1}$$

or

$$y_o = y_s - (z_o - z_s)\tan\theta_y \tag{2}$$

Likewise, for the x-position

$$x_o = x_s - (z_o - z_s)\tan\theta_x \tag{3}$$

where the symbols are as shown in Figures 9a and 9b.

The location of points y_o and x_o on the sensor can be determined using definitions for $\tan\alpha$ and $\tan\beta$ (or similar triangles) as

$$y_o^i = y_o \frac{F}{z_o} \tag{4}$$

and

$$x_o^i = x_o \frac{F}{z_o} \tag{5}$$

where superscript i refers to the position on the image sensor shown in Figures 9, and F is the distance from the receiving lens to the sensor, which is nearly equal to the focal length of the lens for large z_o.

By substituting relationships for y_o and x_o given by Eqs. 2 and 3, the above equations can be rearranged as

$$y_o^i = \frac{F}{z_o}(y_s + z_s\tan\theta_y) - F\tan\theta_y \tag{6}$$

and

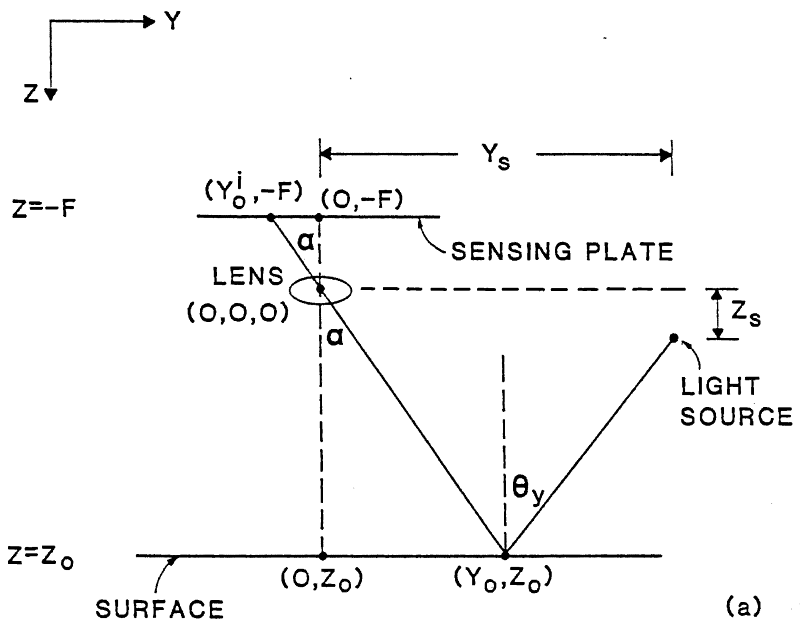


Figure 9a. Two Dimensional View of y-z Plane

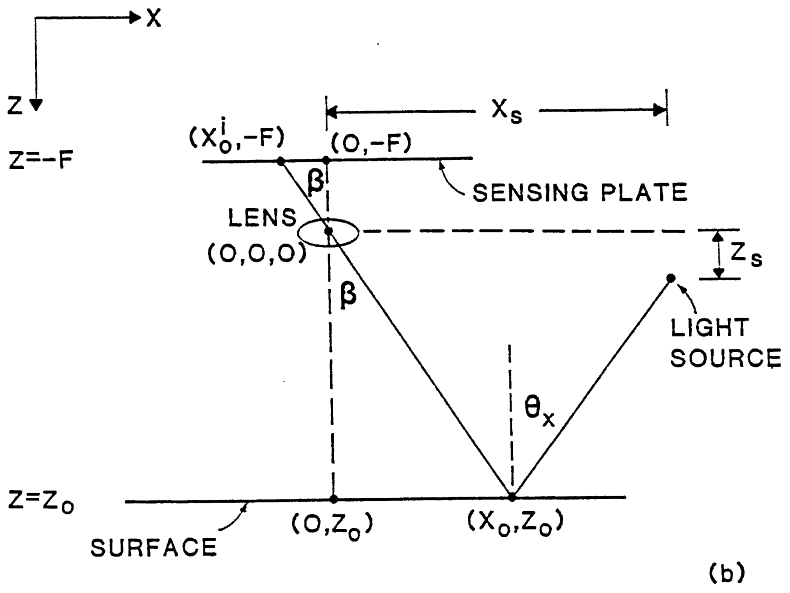


Figure 9b. Two Dimensional View of x-z Plane

$$x_o^i = \frac{F}{z_o}(x_s + z_s \tan\theta_x) - F \tan\theta_x \tag{7}$$

The above equations can be used to determine the image position for any (x_o,y_o,z_o) point on the surface of a projected laser line. Of greater interest in our study is the difference in height relative to some reference elevation. It can be seen from Figure 8, for a line parallel to the y-axis, that a difference in height corresponds to a shift in the pixels of the x-direction and is therefore of primary interest. Pixel differences at some elevation $z=z_o$ relative to a reference elevation of $z=z_{ref}$ can be evaluated as

$$\Delta x_o^i = x_o^i - x_{ref}^i \tag{8}$$

where x_o^i and x_{ref}^i can each be evaluated using Eq. 7 to obtain

$$\Delta x_o^i = F(x_s + z_s \tan\theta_x)\left[\frac{1}{z_o} - \frac{1}{z_{ref}}\right] \tag{9}$$

where Δx_o^i is the pixel difference as shown in the inset of Figure 8.

In measuring soil topography, our objective is to determine $z=z_o$ for a measured Δx_o^i. Therefore, Eq. 9 is rearranged as

$$z_o = \frac{z_{ref}}{1 + \frac{\Delta x_o^i z_{ref}}{K_z}} \tag{10}$$

where

$$K_z = F(x_s + z_s \tan\theta_x) \tag{11}$$

where K_z is a constant for a fixed laser source and for a projected line that is parallel to the y-axis. If z_{ref}, K_z and Δx_o^i are known, the elevation of a point can be determined by Eq. 11. Calibration procedures to determine K_z and z_{ref} are discussed in Chapter V.

Profile Measurement Software

An important step in the profile measurement algorithm is identifying the location of pixels corresponding to the line projected by the laser shown in Figure 8. Since the source is infrared, some background lighting can be removed with an optical filter. Additional filtering is done with software as describe below. Differences in elevations are obtained using the midpoints of the laser line.

The intensity of the laser light across the width of the line follows a Gaussian distribution which is symmetrical about the midpoint, with the brightest pixel located in the center. Although several methods can be used to locate the line, the one chosen for this project involves edge detection/enhancement of the line. Edge enhancement is implemented with spatial filter to increase the contrast between the laser line and the background. The filter is a 1x9 convolution filter selected to use the slope of pixel brightness to enhance the edge of the laser line. A 1x9 filter, rather than a more common 3x3 filter, is used to remove the effects of values above and below the center pixel. The specific values used for this filter are [-2,-2,1,2,2,2,1,-2,-2]. These values were determined by trial-and-error.

After the original image is enhanced, a gray level threshold value is selected to remove background noise. Since the laser light corresponds to the brightest pixels in the image, a histogram of the image can be used to determine the appropriate threshold level. The width of the line appears to be between 4 and 9 pixels for our camera and lens and apparently varies with the intensity of the background lighting. Using 4 to 9 pixels, the appropriate gray level for thresholding would correspond to a value between 98.2% and 99.2% of the cumulative distribution of pixels. For the very low background lighting used in the tests described in this paper, a threshold gray level corresponding to 99% is used.

The final step in finding the midpoint is to determine the edges of the line. This is done by scanning from left to right and locating the first non-black pixel. Once this pixel has been found, the program looks both ways to find the first black pixels on either side of the non-black pixel. Both edges of the line can now be determined. Since the light intensity is symmetrical, the center of the line is found by averaging the two edge values. This center point is the x-direction value. The y direction value is determined by the row number of interest.

The above steps describe the procedures used to determine the midpoint of a single scanning line. The algorithm repeats the process for the 300 most centered lines in an image (each image has 512 lines). The complete image can be analyzed in less than 6 seconds.

Velocity Software

Connectivity Analysis

Velocity is measured by the displacement distance of small wooden beads with time. This requires that the location of beads within the image frame be determined at various points in time. The connectivity analysis given by Cunningham (1981), is the basis for identifying these locations.

In Cunningham's (1981) analysis, the image is first partitioned into regions or "blobs" that correspond to objects, holes, or background in the scene. A blob is simply defined as a connected cluster of pixels which are the same color (i.e. black or white). Since a picture may be composed of a number of blobs, including one for the background, it can be most easily represented as a linked list of blob descriptors. Descriptors are records that contain information about a blob, such as its area, centroid, perimeter, number, gray level and possibly other shape information.

A blob is obtained by determining whether a pixel is "connected" to its adjacent pixels. Adjacent pixels are connected if they are the same grey level.

There are several conventions in a rectangular grid to specify which pixels are adjacent, but for this research, a 6-connectivity convention was used.

Run-length encoding is used to determine the blob number of a previously processed pixel. In the coding scheme, each unbroken run of 0's or 1's in a raster line of the image is described by a record that tells its starting (leftmost) column, its length, and blob number. The entire line is then described by a list of these run-length records which are allocated sequentially as the image is scanned.

The connectivity analysis algorithm scans the image from left to right, top to bottom, updating the descriptors of each blob which intersect the current scan line. At the end of each run of 0's or 1's, the run-length list is updated and blob statistics are accumulated. If any pixel of the run just completed is 6-connected to a pixel of the same gray level on the previous line, an existing blob is extended to include this information. Otherwise, a new blob record is allocated. Each blob is assigned a unique number. Pixels belonging to the blob are labeled with this number in the run-length list.

Velocity Algorithm

The connectivity analysis and the DT-IRIS functions are used to determine velocity. Beads are first moved under the camera as a result of their velocities. The camera then "snaps" an image and sends a video signal to the computer where it is digitized using the Data Translation frame grabber board. The frame rate of the camera is one frame per 1/30 of a second. It is also possible for the camera to send a signal to a video cassette recorder to avoid real time processing. The VCR unit has the same frame rate as the camera.

The video signal from the camera or the VCR unit is digitized into various gray levels by the frame grabber board. A 2x2 low pass convolution is used to filter out high frequency noise. A threshold gray level is then chosen by varying the threshold value until blobs best depict the true size of the beads. This threshold value is held constant for all velocity measurements, which is a valid assumption for unaltered lighting conditions.

After the threshold value has been selected, the user visually observes the beads as they flow past the camera and selects a frame for processing. This frame is stored in one of the on-board frame memory buffers, and a second frame is snapped three frames later. The second frame is stored in the other on-board frame buffer.

For both frames, the low pass convolution filter and the threshold value are used to obtain binary images. The connectivity analysis is then done individually for the blobs (in this case the blobs are beads) in each frame, and blob statistics are determined. The user visually matches the blobs in the first frame with the appropriate blobs in the second frame. The displacement distance is defined as the distance between the centroids of each pair of blobs. The user can disregard any blobs that are in one frame but not in the next.

The distance between the centroids of the blobs is converted from pixels to inches using a conversion factor which is determined in a calibration procedure described in Chapter V. Time lapse between images is calculated using the number of frames between snaps and the frame rate of 1/30 of a second. The velocity is simply the displacement distance divided by the time lapse.

CHAPTER V

EXPERIMENTAL PROCEDURES

Introduction

The testing of the system and algorithms for soil profile and velocity measurements is done in two steps. First, the system is calibrated. After calibration, experiments to test the system hardware and software are performed. These tests are used to assess the capabilities of the image processing system. Calibration and experimental procedures are both described in this chapter.

Calibration Procedures

General

Algorithms for soil profile and velocity measurements both require distance values in length units such as inches or millimeters. The profile algorithm uses the distance between the laser light line and a reference line, and the velocity measurement algorithm uses the distance between the centroids of blobs. The image processing system inherently works with units in pixels, instead of more meaningful length units. Calibration procedures to convert pixel distances to length measurements are discussed in this section.

Conversion factors are required for the x, y and z directions for the soil profile measurements and the x and y directions for the velocity measurements. Calibration in the z-direction requires the determination of K_z and z_{ref} given in Eq.

10. It is described separately. Calibration procedures in the x and y directions are very similar and are discussed together. These procedures are conducted prior to the start of each run.

Vertical Direction

Two calibration factors, K_z and z_{ref}, are needed to calculate depth using Eq. 10. The first step in the calibration procedures is to move the structured lighting equipment over a horizontal plate which is by definition at the reference elevation (i.e., z_{ref}). Two blocks of known heights are placed on the plate. The elevation at the top of these blocks can then be defined as (downward positive)

$$z_1 = z_{ref} - h_1 \tag{12}$$

and

$$z_2 = z_{ref} - h_2 \tag{13}$$

where z_1 and z_2 are elevations of the top of the first and second blocks, respectively, and h_1 and h_2 are the heights (known) of these two blocks.

Changes in pixel location between the reference plate and the top of each block can be defined directly from Eq. 9, or,

$$\Delta x_1^i = K_z \left[\frac{1}{z_1} - \frac{1}{z_{ref}} \right] \tag{14}$$

and

$$\Delta x_2^i = K_z \left[\frac{1}{z_2} - \frac{1}{z_{ref}} \right] \tag{15}$$

where Δx_1^i and Δx_2^i are the differences in pixel locations from the reference elevation to the elevations of the top of the first and second blocks. These differences can be determined by software and thus are known values. The definition of K_z given by Eq. 11 is used in the above equations.

Using Eqs. 12 and 13, Eqs. 14 and 15 can be written as

$$\Delta x_1^i = K_z \left[\frac{1}{z_{ref} - h_1} - \frac{1}{z_{ref}} \right] \tag{16}$$

and

$$\Delta x_2^i = K_z \left[\frac{1}{z_{ref} - h_2} - \frac{1}{z_{ref}} \right] \tag{17}$$

We now have two equations (i.e., Eqs. 16 and 17) and two unknowns (K_z and z_{ref}). The ratio of the above equations can be written as

$$\frac{\Delta x_1^i}{\Delta x_2^i} = \frac{h_1(z_{ref} - h_2)}{h_2(z_{ref} - h_1)} \tag{18}$$

To simplify typography a dimensionless variable v is defined as

$$v = \frac{\Delta x_1^i / h_1}{\Delta x_2^i / h_2} \tag{19}$$

so that z_{ref} is calculated as

$$z_{ref} = \frac{v h_1 - h_2}{v - 1} \tag{20}$$

The value for K_z can now be determined from Eqs. 16 or 17. Using Eq. 16, K_z is calculated as

$$K_z = \frac{\Delta x_1^i}{\frac{1}{z_{ref} - h_1} - \frac{1}{z_{ref}}} \tag{21}$$

K_z indirectly incorporates the conversion of pixels to a length measurement. After K_z and z_{ref} are determined, the elevation of a reflected surface can be calculated directly from Eq. 10.

The algorithm to locate the laser light uses 300 of the most centered lines of an image. The solution of Eqs. 17 and 18 could then be obtained for each one of these 300 lines. Different values for K_z and z_{ref} are possible because of slight imperfections in the construction of mechanical components and possibly other factors. Scatter in values was avoided by fitting a least squares line to the observed points and by using the intercept values for estimating Δx_1^i and Δx_2^i. Corrections are also included to account for a laser line that is not perfectly parallel to the y-axis.

Horizontal Plane (x and y directions)

Conversion factors are also needed to convert pixel values to length measurements in the x and y directions. These factors are determined without using the laser light. A card with two sets of parallel lines, one in the x direction and one in the y direction, is viewed from the camera. Both set lines are located by software. A least squares line is again fitted through points to account for slight imperfections. Distance in pixels between the parallel lines in the x and y directions is determined using the midpoints of the least squares lines. The conversion factors are then calculated as the ratio of known distances between parallel lines and measured pixel values. This calibration procedure is done at an elevation of z_{ref}.

Conversion factors are a function of the height of the surface being measured. This concept is illustrated in Figure 10 for the y direction. A similar figure could be drawn for the x direction. The origin is again at the lens. At the reference height, the conversion factors described in the previous paragraph can be used to determine the distance measurement between $(0,z_{ref})$ and (y_{ref},z_{ref}) from a measured pixel difference Δy^i as

$$y_{ref} = K_{y,ref}\Delta y^i \tag{22}$$

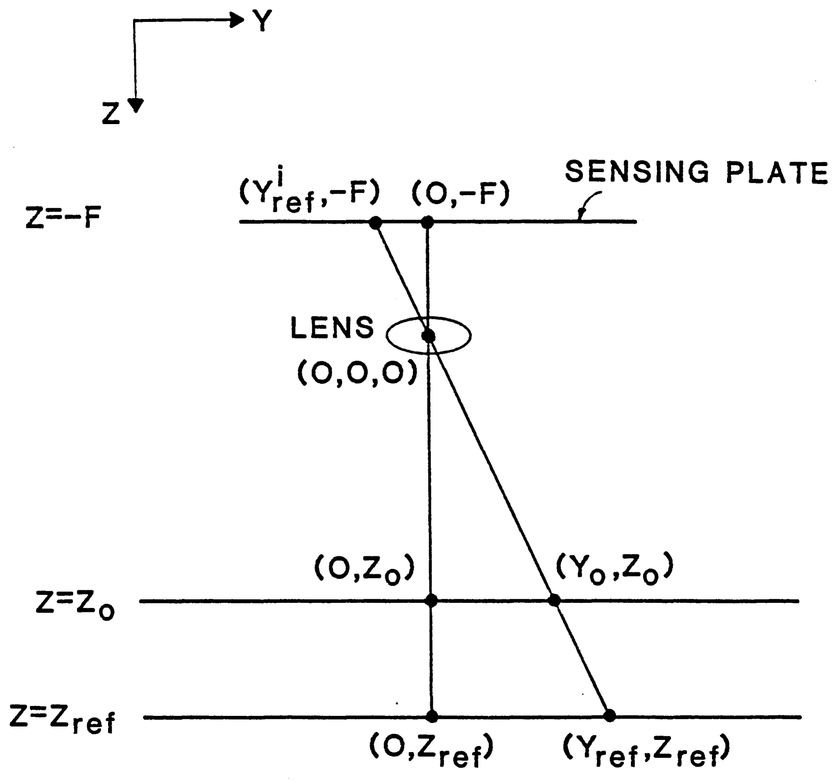


Figure 10. Schematic Illustrating Change in K_y with Elevation

where $K_{y,ref}$ is the conversion factor from pixel-to-distance measurements for the y direction at the reference height and other terms are as defined in Figure 10.

In Figure 10, consider the situation when the height of the surface has been moved to $z=z_o$. The pixel difference Δy^i is constant, but the distance measurement is now the difference between $(0,z_o)$ and (y_o,z_o). Using similar triangles, we can write

$$y_{ref} = y_o \frac{z_{ref}}{z_o} \tag{23}$$

which can be substituted into Eq. 22 to obtain,

$$y_o = \left(K_{y,ref} \frac{z_o}{z_{ref}} \right) \Delta y^i \tag{24}$$

Equation 24 is used to calculate distance measurement for a measured pixel difference at elevations different than the reference height. It essentially adjusts the reference height conversion factor by the ratio of the surface and reference elevations.

Likewise, for the x direction one would obtain

$$x_o = \left(K_{x,ref} \frac{z_o}{z_{ref}} \right) \Delta x^i \tag{25}$$

where $K_{x,ref}$ is the conversion factor from pixels-to-length measurements for the x direction at the reference height, Δx^i is the pixel difference in the x direction (from zero) and other terms are as previously defined.

Testing Procedures

Surface Profile Measurements

General. Two sets of experiments were conducted to evaluate the accuracy and applicability of structured lighting techniques for measuring soil surface profiles. The first set of experiments used "precision", rigid objects. These objects were constructed precisely and represented well-defined shapes for evaluating accuracy. The second set of experiments used soil with pre-formed surface depressions to test the applicability of the system.

The first step in the testing procedures was to calibrate the conversion factors discussed in the previous section. Stepper motors were used to move the structured lighting equipment as described earlier. Adjustment in the coordinate system was obtained from the number of steps and the distance traveled per step. Location of the laser line was determined by the algorithm previously described in Chapter IV. Because of the large amount of data that can be gathered by the system, separate programs were written to encode and store the data in a compressed format and to conduct subsequent analyses.

Rigid Objects. Two different sets of blocks were used for the rigid object testing. The first set consisted of five square tubing blocks, each of a different height. The second set of blocks was made from solid square steel with milled trapezoidal surface notches of varying dimensions. Measurements of block dimensions were obtained using calipers with an accuracy of roughly 0.05 mm. The square tubing blocks were used to evaluate the accuracy of height measurements. The trapezoidal notched blocks were used to evaluate the accuracy of measured channel geometries. A schematic showing the dimensions of interest in the trapezoidal notched blocks is shown in Figure 11.

Each block was measured eight times by the image processing system to check

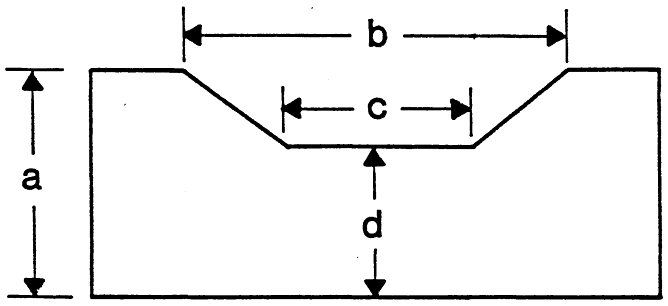


Figure 11. Symbols of Trapezoidal Notches Used in Tables III and IV

the system for repeatability and accuracy. In all tests, the blocks were placed on the flat plate used to calibrate the system. The stepper motors were therefore not used to move the equipment in the x and y directions. Several tests, however, were made to check the stepper motor routines and evaluate the effects of moving the camera system vertically.

The first four tests are similar to that illustrated in Figure 8 where the reference elevation and surface elevation of the block are in the same image. Block heights can then be evaluated directly by the pixel difference as illustrated in the insert of Figure 8. The only differences between the first four tests are due to the movement of the camera system vertically. Tests #5 through #8, however, store the baseline reference elevation in a data file at calibration. Elevations of the block surfaces are then determined from separate images and subtracted from the stored reference elevation to estimate block height. Differences in these four tests are again due to the movement of the camera system vertically. A summary of the conditions for each test is given below:

1. Camera equipment was left in the same position as it was for calibration,
2. Camera was moved up 500 steps and then down 500 steps (roughly 64 mm) to test stepping motor routines,
3. Camera equipment was moved up 500 steps to account for changes in conversion factors with camera height,
4. Camera equipment was lowered 250 steps (roughly 32 mm) from its original position to account for changes in conversion factors and to provide a larger vertical range on measured values,
5. Repeat Test #1 using reference elevation stored in data file,
6. Repeat Test #2 using reference elevation stored in data file,
7. Repeat Test #3 using reference elevation stored in data file,
8. Repeat Test #4 using reference elevation stored in data file.

Soil Profile. The ability of the structured lighting system to measure soil profiles was examined using two different soil types. A sandy soil and a loam soil were placed in two large trays. A varying surface profile was formed in each soil. This profile was measured using both the structured lighting system and a pin displacement unit. A 0.3 m (1 foot) section of the profile was measured. The spacing of the pins was 12.7 mm (0.5 inches).

A disadvantage of evaluating the structured lighting system using soil profiles is the limited accuracy of determining the actual elevation of the soil. The accuracy of our pin displacement unit was estimated as roughly $\pm$ 2 mm. This value was obtained by taking several readings at the same site with different people.

In this set of experiments, the structured lighting system was calibrated and then moved by stepper motors to the soil trays. Measurements were then taken in the region of interest. Elevations of the soil profile were gathered relative to the elevation of the calibration plate. The pin displacement unit determines the variations in height (z direction) with distance in the y direction for a given x value. The structured lighting system measures the variation in height in the y direction, but the value for x varies with height as previously discussed. Therefore, it was not possible to match pin locations and structured lighting readings exactly. Reasonably close values could be obtained by making several passes of the structured lighting system surrounding the pin locations and by selecting x locations that were closest to the pin values.

Velocity Measurements

Two testing procedures were conducted to evaluate the velocity measurement system. The first test was used to determine differences between measured velocities from a live camera and those obtained using a VCR playback unit. The second set of tests was done to evaluate the accuracy of the algorithm.

Both testing procedures used the rotational speed of a conveyor belt driven by a variable speed motor to move the beads. Painted beads were glued randomly to the belt. The camera was mounted in a stationary position over the belt and was calibrated for this position. The actual velocities of the beads were determined using a wheel tachometer and converting measured rpm values to linear velocities (m/s). The accuracy of the tachometer was roughly 0.009 m/sec.

Ten different speeds were tested using the live camera, and ten different speeds were recorded onto video tape. The varying speeds were then played back and the algorithms were used to measure the velocity of the beads on tape. Although the speeds for live and taped velocities were different, comparisons are still possible by using differences between belt velocities and those measured.

The precision of the velocity measurements was evaluated by repeating the taped velocities. Small discrepancies between measurements are possible, because each frame may have different beads. For each belt velocity, the bead velocity was measured eight times at varying points in time. The number of beads used in each measurement was then counted and an overall average velocity was determined. Standard deviation of the velocities was also determined for each speed.

CHAPTER VI

RESULTS AND DISCUSSION

Introduction

The results of the experimental testing are discussed in two sections. In the first section, the accuracy and applicability of the soil profile measuring system are evaluated. This section also includes an application of the system to a three-dimensional surface. The second section is used to present and discuss the results obtained with the velocity measuring system.

Profile Measurements

Rigid Objects

A print of an image produced with structured lighting is shown in Figures 12a and 12b. Figure 12a shows a trapezoidal notched block used in the experiments. The corresponding image seen by the structured lighting system is shown in Figure 12b. These figures clearly show the displacement of pixels in the x-direction caused by a difference in surface elevations.

The results obtained for the five square tubing blocks are shown in Table I for all eight tests. Data shown in this table also incorporate possible errors in the vertical movement of the structured lighting equipment by the stepper motor. No definite trends can be seen with the results shown in Table I. The maximum difference in measured values for a given block was only 0.63 mm and a maximum

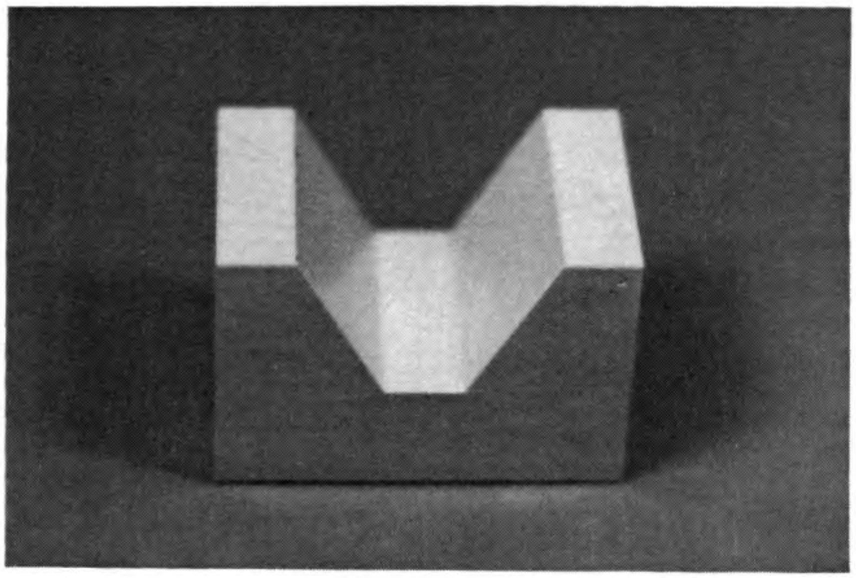


Figure 12a. View of a Trapezoidal Notched Block

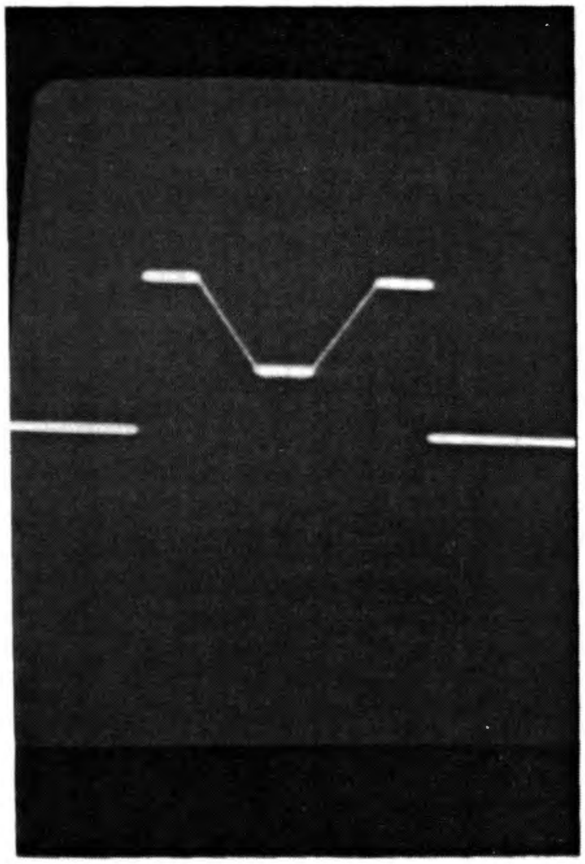


Figure 12b. Corresponding Screen Image of Laser Line

TABLE I

ACTUAL AND MEASURED BLOCK HEIGHTS OF RECTANGULAR BLOCKS

Test ID	Block #1 (mm)	Block #2 (mm)	Block #3 (mm)	Block #4 (mm)	Block #5 (mm)
Actual*	12.80	25.04	37.69	50.37	62.08
#1	12.80	24.99	37.79	50.44	62.08
#2	12.95	25.04	37.64	50.29	62.03
#3	12.73	24.77	37.16	50.16	61.98
#4	12.75	24.84	37.69	+++	+++
#5	12.83	24.79	37.52	50.19	61.98
#6	13.00	25.10	37.72	50.44	62.13
#7	13.00	25.04	37.69	+++	+++
#8	12.67	25.02	37.59	50.32	61.90
Max. Diff.	0.33	0.33	0.63	0.28	0.23

*Actual value represents the best estimate possible using using instruments and equipment with limited precision.

+++Laser light was outside the view of the camera.

difference for most blocks of roughly 0.30 mm. A summary of these measurements is given in Table II, using mean values and standard deviations. The mean value is within 0.1 mm of the actual value for each block height.

The results for trapezoidal notched blocks are given in Table III and Table IV. Table III shows the height dimensions of the block and Table IV shows the width dimensions. Only mean and standard deviation values for the eight tests on these blocks are presented. Measurement accuracy for the geometry of these blocks is roughly equivalent to that obtained for the square tubing. The mean value for each geometric characteristic is within 0.15 mm of the actual value.

The results of the rigid block tests indicate that structured lighting techniques are capable of measuring geometric characteristics with good accuracy. Maximum errors in measuring these characteristics are less than 1 mm for all runs. Differences between actual and measured mean values are less than 0.15 mm.

Soil Profile

The comparison of the soil profiles measured using the structured lighting system and the pin displacement unit for the sandy soil and loam soil are shown in Figure 13 and Figure 14, respectively. The uncertainty region ($\pm$ 2 mm) of the pin displacement unit are shown around each point. As previously discussed, the results of the structured lighting technique gives the x, y, and z coordinates of a point. The plotted laser lines are the y and z values calculated for the 0.3 m section for each soil type. Approximately 600 data points were used to construct each laser line.

As shown by Figure 13 and Figure 14, the structured lighting technique values are within the accuracy of the pin displacement values. The trend of the pins in the sandy soil to be lower than the laser line is probably caused by pins sinking into

TABLE II

SUMMARY OF HEIGHT MEASUREMENTS

Block	Actual Height (mm)	Mean Value (mm)	Std Dev. (mm)
#1	12.80	12.84	0.128
#2	25.04	24.95	0.128
#3	37.69	37.60	0.196
#4	50.37	50.31	0.119
#5	62.08	62.02	0.082

TABLE III

ACTUAL AND MEASURED HEIGHT DIMENSIONS
OF TRAPEZOIDAL NOTCHED BLOCKS

	Actual dimension		Mean Value		Std. Dev.	
	a (mm)	d (mm)	a (mm)	d (mm)	a (mm)	d (mm)
Block #6	44.45	19.02	44.34	19.02	0.180	0.056
Block #7	44.32	25.22	44.27	25.15	0.168	0.113
Block #8	44.32	31.77	44.27	31.67	0.164	0.109

TABLE IV

ACTUAL AND MEASURED WIDTH DIMENSIONS
OF TRAPEZOIDAL NOTCHED BLOCKS

	Actual dimension		Mean Value		Std. Dev.	
	b (mm)	c (mm)	b (mm)	c (mm)	b (mm)	c (mm)
Block #6	41.80	12.62	41.78	12.75	0.187	0.250
Block #7	47.47	25.04	47.45	25.12	0.171	0.174
Block #8	51.90	37.63	51.97	37.64	0.026	0.038

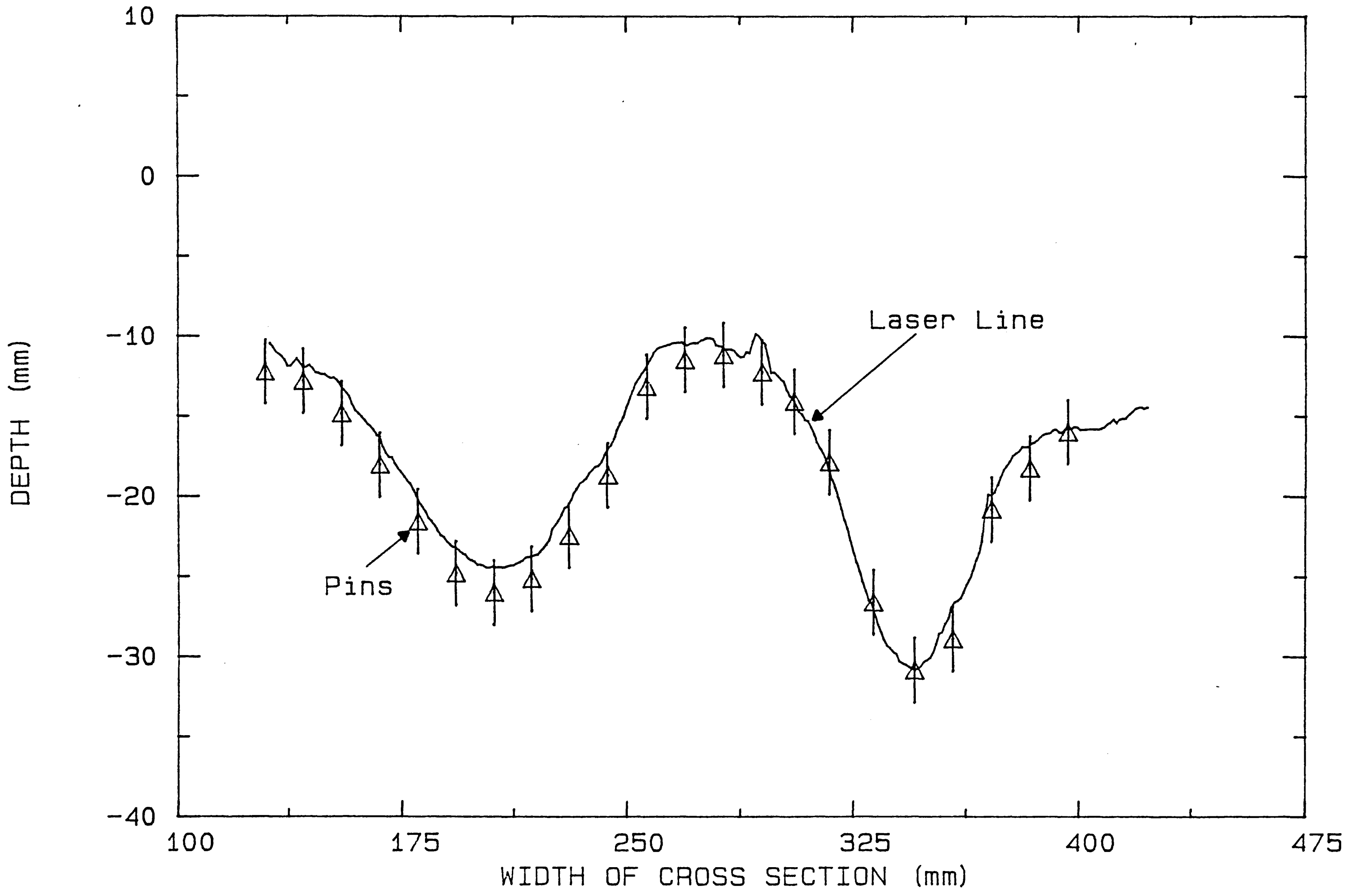


Figure 13. Soil Profile for Sandy Soil

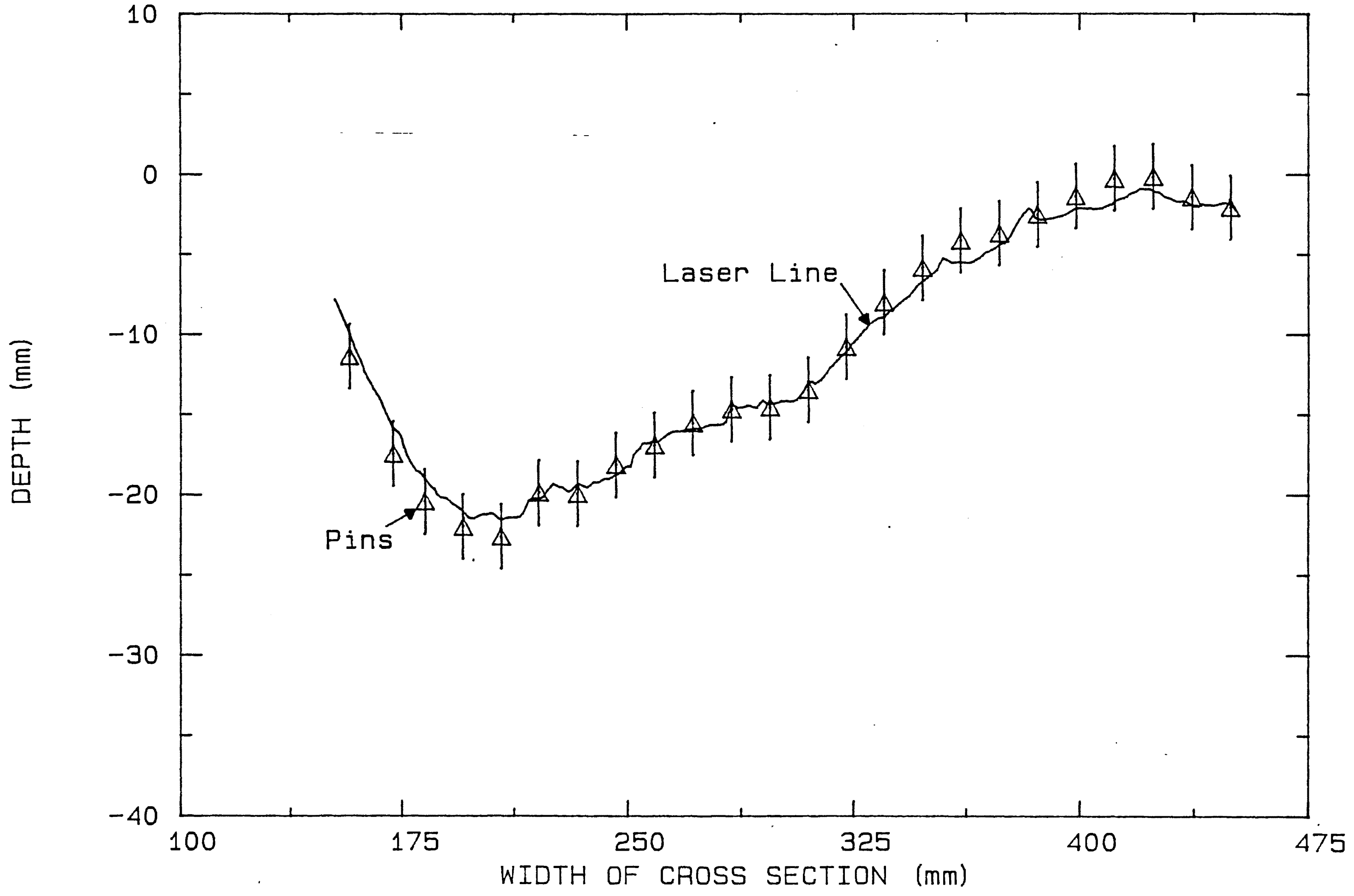


Figure 14. Soil Profile for Loam Soil

the surface. Therefore, the measured values of the pins tend to be lower than the laser values.

The results of the rigid object and soil profile tests indicate that the structured lighting system is an alternative technique for measuring soil profiles. Additional runs have been made with wet and dry soils. No noticeable problems have been observed as long as water is not ponded on the surface.

Applications

Results previously discussed are limited to measuring a single profile. Additional algorithms have been written to analyze the data gathered with several passes of the system. Three-dimensional views of the soil surface can then be obtained for a given soil area. The algorithms have been specifically written for use with the erosion table described in Chapter III, but could be easily modified for field work.

A three-dimensional analysis of the soil surface is obtained using the stepper motors and software previously described. Successive measurements of a specified area are first made in the erosion table. This information is stored in encoded form. Data points are then analyzed to determine average height values for a specific grid. These results can be displayed graphically with two figures shown simultaneously on the screen. In Figure 15, an example of the two figures for a 0.3 by 0.3 m area of the erosion table is shown. The first figure is a topographical view (x and y values) of varying gray levels. Each gray level represents a different height. As shown in Figure 15, a three-dimensional view of a rill can be seen from this representation. The bottom figure shows the cross-section of a specified slice of the three-dimensional view. The location of the slice is shown in the top figure. This slice can be rotated so that various cross-sections can be seen.

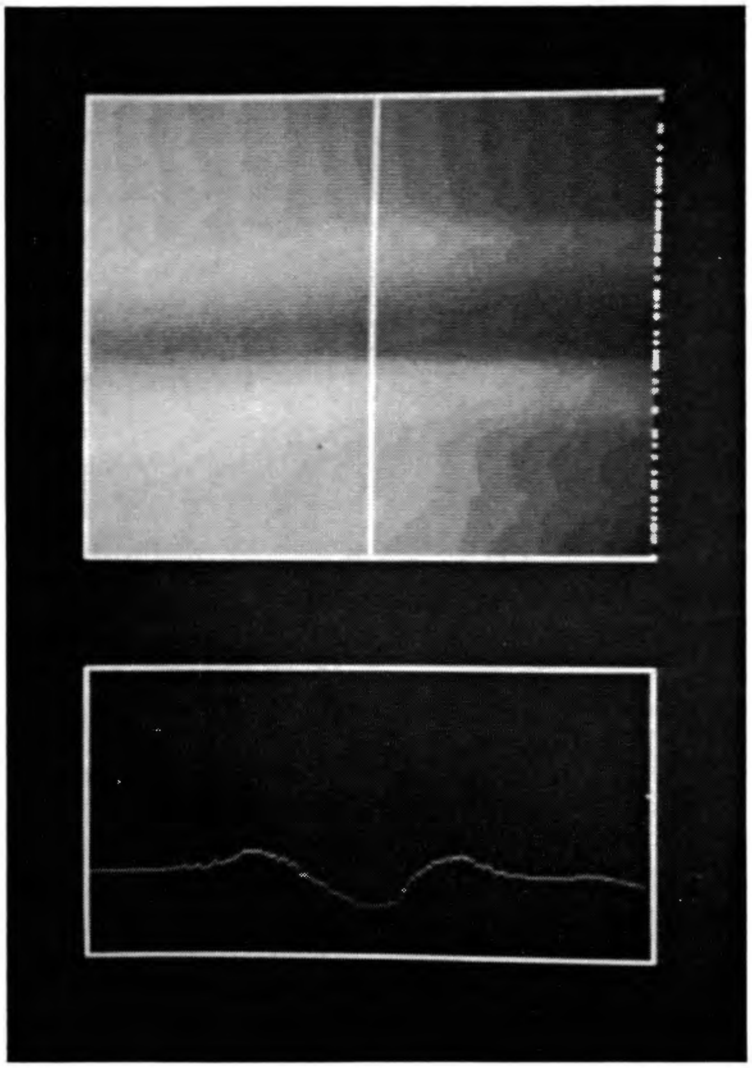


Figure 15. Topographical and Cross-sectional View of Soil Surface Area

Velocity Measurements

Comparison Test

A print of the final image of the velocity measuring algorithm is shown in Figure 16. The double circles represent the location of the beads which were snapped in the first frame. The blobs are the beads of the second snapped frame. The line between the first and second frame beads is the displacement of the centroids which results in the distance the bead has traveled.

The results of the comparison between live measurements and taped measurements are shown in Table V. Although the actual velocities are different for the two types of measurements, a comparison can still be made by considering the accuracy of the velocity routines. The maximum difference using live processing is 0.006 m/s and is 0.009 m/s using the recorded images. Both of these values fall within the measurement accuracy of the actual values.

Results of this test show that there is no substantial difference for using the velocity system between live velocities and recorded velocities. This allows various flows to be recorded first and then analyzed at a more convenient time.

Live data in Table V were also used to estimate the maximum velocity measurable with this particular algorithm. The maximum value was approximately 0.914 m/s (3 ft/sec). At larger velocities, the beads are traveling so fast that they are not in successive frames. A fast velocity routine needs to be developed to measure speeds greater than 0.914 m/s.

Accuracy Test

The accuracy and precision of the velocity measuring system were determined by measuring the same speed eight different times. The results of these tests are shown in Table VI. Since the velocities were pre-recorded, it was possible to

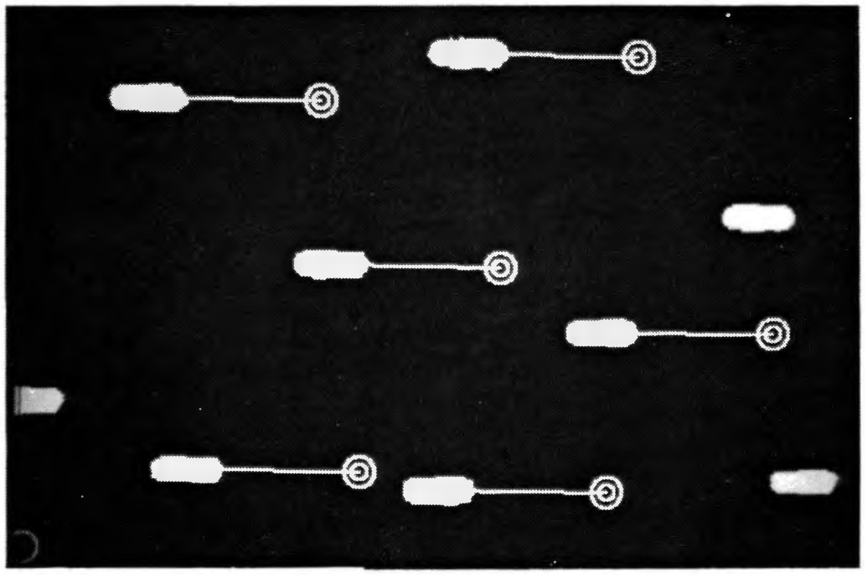


Figure 16. Displacement of Beads for Velocity Algorithm

TABLE V

COMPARISON OF LIVE AND RECORDED VELOCITIES

Live		Recorded	
Actual Velocity (m/s)	Measured Velocity (m/s)	Actual Velocity (m/s)	Measured Velocity (m/s)
0.411	0.416	0.360	0.360
0.457	0.460	0.460	0.460
0.478	0.478	0.527	0.521
0.610	0.616	0.567	0.570
0.668	0.668	0.582	0.579
0.686	0.692	0.643	0.634
0.734	0.734	0.725	0.722
0.817	0.820	0.741	0.732
0.863	0.863	0.811	0.811
0.914	0.911	0.875	0.869

TABLE VI

ACTUAL AND MEASURED VELOCITIES (m/s)

Run #	True Speed ($\pm$0.009)	Number of Blobs per frame	Average Max. Dev. per frame	Average Mean Velocity	Standard Error
1	0.340	5 to 7	0.004	0.340	0.005
2	0.460	4 to 6	0.004	0.463	0.009
3	0.570	4 to 6	0.004	0.570	0.004
4	0.579	4 to 6	0.007	0.579	0.003
5	0.634	4 to 5	0.009	0.637	0.002
6	0.722	3 to 4	0.006	0.725	0.002
7	0.811	2 to 4	0.006	0.808	0.003
8	0.869	2 to 3	0.009	0.869	0.004

measure speeds at varying points on the tape, resulting in a different number of beads in each velocity calculation. A velocity for each bead was first calculated. The average of all beads in a frame was used to calculate a mean velocity, and the average of this value for eight frames was used to calculate the average mean velocity.

The variability in bead speeds for individual frames is summarized by the average maximum deviation per frame in Table VI. These values were obtained by averaging the maximum deviation observed for the eight different frames. The largest average maximum deviation was only 0.009 m/s, indicating small variations between individual bead speeds for a given frame.

The average mean velocity was obtained by averaging the mean velocity for eight different frames. As shown in Table VI, the average mean velocities were within the uncertainty of true speed values. The variability in mean velocities between frames is reflected in the standard errors given in Table VI. These values represent the standard deviation of the mean velocities about the average mean velocity. The small standard errors show that the system is consistent in its measured values.

These results indicate that the velocity measuring system is an acceptable procedure for measuring the velocity of beads. The accuracy of the system is within $\pm$ 10 mm/s. Additional tests are needed with flowing water before the technique can be applied to rill flows.

CHAPTER VII

SUMMARY AND CONCLUSIONS

A study was conducted to quantify erosion processes using a low cost image processing system. The objectives of the study were (1) to design and develop an apparatus to measure soil profiles and water velocity using image processing techniques, (2) to test the system for measuring soil profiles using well-defined objects of known size and shapes and using actual soil profiles, and (3) to test the system for measuring water velocities using known velocities of small wooden beads.

The image processing measuring system was incorporated into a large-scale laboratory apparatus. The mechanical, electrical and structural components of the measurement system, including an erosion table, soil profile measuring equipment, and velocity measuring equipment were described. The system is controlled by an IBM-AT. Data Translation's DT-2851 frame grabber and DT-2858 co-processor boards are used to digitize and manipulate images.

Surface topography measurements are made using a structured lighting technique. This technique requires that a well-defined stripe of light be projected onto the surface by a laser. Equations to determine elevation and the location of the stripe within the field of view of a camera are developed. Calibration procedures are also discussed.

Image processing techniques to measure surface velocities are developed using the travel distance of small wooden beads in a specified time period. Travel distance is obtained by analyzing two frames at different points in time. Contrast between beads and background is enhanced by painting the beads with flourescent paint and using an ultra-violet bulb for a light source. The computational speed of the velocity measurement algorithm is too slow to run in real time. Therefore, the movement of the beads is first taped using a NEC high quality digital video cassette recorder.

Two sets of experiments were conducted to evaluate the accuracy and applicability of structured lignting techniques for measuring soil surface profiles. The first set of experiments used rigid objects. These objects were constructed precisely and represented well-defined shapes for evaluating accuracy. The second set of experiments used soil with pre-formed surface depressions to test the applicability of the system.

Two testing procedures were conducted to evaluate the accuracy of the velocity measurement system. The first tests were used to determine differences between measured velocities from a live camera and those obtained using a VCR playback unit. The second set of tests was done to evaluate the accuracy of the algorithm.

Based on the experimental results, the following conclusions may be drawn.

1. The results of the rigid block tests indicate that structured lighting techniques are capable of measuring geometric characteristics with good accuracy. Errors in measuring these characteristics were less than 1 mm for all runs.

2. The structured lighting is applicable to soil profiles. No noticeable problems have been observed as long as water is not ponded on the surface.

3. There seems to be no substantial differences between live velocities and those recorded on video tape.

4. The maximum velocity for which this particular algorithm can be utilized is approximately 0.914 m/s (3 ft/sec).

5. The velocity measuring system appeared to be a useful technique for measuring surface velocities. The accuracy of the system is at least $\pm$ 10 mm/s.

Recommendations for Future Research

1. Perform surface measurements on actual rills in laboratory and field settings.
2. Develop a rain/light shield for the camera and laser to protect them from water and to limit outside lighting for consistent thresholding.
3. Design a portable system for field work.
4. Test the slow water algorithm with open channel flows.
5. Develop a fast velocity algorithm to measure accurately water velocities greater than 0.914 m/s.

REFERENCES CITED

Albertson, M. L., J. R. Barton and D. B. Simons. 1960. Fluid Mechanics for Engineers. Prentice-Hall, Inc. New Jersey.

Barfield, B. J. and W. H. Henson, Jr. 1971. Calibration of hot-wire and hot-film probes. Transactions of the ASAE. 14(6):1100-1102, 1106.

Barfield, B. J., E. A. Hiler and E. T. Smerdon. 1968. The effects of rainfall on the structure of turbulence in shallow open channel flow. ASAE Paper No. 68-748, ASAE, St. Joseph, MI 49085.

Beasley R. P., J. M. Gregory and T. R. McCarty. 1984. Erosion and Sediment Pollution Control. 4th Ed. Iowa State University Press. Ames.

Bennett, H. H. 1939. Soil Conservation. McGraw-Hill. New York.

Beckwith, T. G., N. L. Buck and R. D. Marangoni. 1982. Mechanical Measurements. Addison-Wesley Publishing Comapany, Inc. Reading, Massachusetts.

Burwell, R. E., R. P. Allmaras and M. Amemiya. 1963. A field measurement of total porosity and surface microrelief of soils. Soil Sci. Soc. Am. Proc. 27:697-700.

Clark II, E. H., J. A. Haverkamp and W. Chapman. 1985. Eroding Soils The Off-Farm Impact. The Conservation Foundation. Washington D.C.

Cunningham, R. 1981. Segmenting binary images. Robotics Age. 3(4):4-19.

Currence, H. D. and W. G. Lovely. 1970. The analysis of soil surface roughness. Transactions of the ASAE. 13(6):710-714.

Curtis, W. R. and W. D. Cole. 1972. Micro-topograghic profile gage. Agricultural Engineering. 53:17.

DeBoodt, M. and D. Gabriels. 1980. Assessment of Erosion. John Wiley & Sons. New York.

DePloey, J. and D. Gabriels. 1980. Measuring soil loss and experimental studies. Soil Erosion. John Wiley & Sons. New York.

Ellison, W. D. 1944. Studies of raindrop erosion. Agricultural Engineering. 25:131-136, 181-182.

Foster, G. R., L. F. Huggins and L. D. Meyer. 1984. A laboratory study of rill hydraulics: I. velocity relationships. Transactions of the ASAE. 27(3):790-796.

Foster, G. R. and L. D. Meyer. 1972. Efficient processing of microrelief photographs. ASAE Paper No. 72-593, ASAE, St. Joseph, MI 49085.

Gilley, J. E., S. C. Finker and G. E. Varvel. 1986. Hydraulic and soil loss variables on eroding area. ASAE Paper No. 86-2041, ASAE, St. Joseph, MI 49085.

Grant K. E. 1975. Erosion in 1973-74: The record and the challenge. J. Soil Water Cons. 30:29-32.

Harral, B. B. and C. A. Cove. 1982. Development of an optical displacement transducer for the measurement of soil surface profiles. Jour. Agric. Engr. Research. 27:421-429.

Heermann, D. F., R. J. Wenstrom and N. A. Evans. 1969. Prediction of flow resistance in furrows from soil roughness. Transactions of the ASAE. 12(4):482-485,489.

Henry, J. E., M. J. Sciarini, and D. M. Van Doren, Jr. 1980. A device for measuring soil surface profiles. Transactions of the ASAE. 23(6):1457-1459.

Hirschi, M. C., B. J. Barfield and I. D. Moore. 1984. Rillmeters for detailed measurement of soil surface heights. ASAE Paper No. 84-2534, ASAE, St. Joseph, MI 49085.

Hudson, N. 1981. Soil Conservation. 2nd Ed. Cornell University Press.

Jalkio, J. A., R. C. Kim and S. K. Case. 1985. Three dimensional inspection using multistripe light. Optical Engineering, Bol. 24(6):966-974.

Khorashahi, J., R. K. Byler, and T. A. Dillaha. 1985. An opt-electronic soil profiler. ASAE Paper No. 85-3041, ASAE, St. Joseph, MI 49085.

Kuipers, H. 1957. A relief meter for soil cultivation studies. Netherlands J. of Agr. Sci. 5:255-262.

Line D. E. and L. D. Meyer. 1987. Flow velocities of concentrated runoff along cropland furrows. ASAE Paper No. 87-2087, ASAE, St. Joseph, MI 49085.

McCool, D. K., M. G. Dossett, and S. J. Yecha. 1976. A portable rill meter for measureing soil loss. ASAE Paper No. 76-2054, ASAE, St. Joseph, MI 49085.

McCool, D. K., M. G. Dossett, and S. J. Yecha. 1981. A portable rill meter for field measurement of soil loss. Proceedings of the International Symposium on Erosion and Sediment Transport Measurement, June, 1981, Florence, Italy.

Merva, G. E., R. D. Brazee, G. O. Schwab and R. B. Curry. 1970. Theoretical considerations of watershed surface description. Transactions of the ASAE. 13(4):462-465.

Mitchell, J. K. and B. A. Jones. 1973. Profile measuring device. Transactions of the ASAE. 16(3):546-547.

Moore, I. D. and C. L. Larson. 1979. Estimating micro-relief surface storage from point data. Transactions of the ASAE. 22(5):1073-1077.

Morgan, R. P. C. 1986. Soil Erosion and Conservation. Longman Group UK Limited. England.

Podmore, T. H. and L. F. Huggins. 1981. An automated profile meter for surface roughness measurements. Transactions of the ASAE. 24(3):663-665, 669.

Radke, J. K., M. A. Otterby, R. A. Young and C. A. Onstad. 1981. A microprocessor automated rillmeter. Transactions of the ASAE. 24(2):401-404, 408.

Rice, C., B. N. Wilson and M. Appleman. 1987. Instrumentation for erosion research using image processing techniques. ASAE Paper No. 87-2097, ASAE St. Joseph, MI 49085.

Richardson, E. V. and R. S. McQuivey. 1968. Measurement of turbulance in water. Journal of the Hydraulics Division. Proc. of the ASCE. 15(3):411-429

Romkens, M. J., S. Singarayar, and C. J. Gantzer. 1982. An automated noncontact surface profile meter. ASAE Paper No. 82-2620, ASAE St. Joseph, MI 49085.

Schafer, R. L. and W. G. Lovely. 1967. A recording soil surface profile meter. Agricultural Engineering. 48:280-282.

Schwab, G. O., R. K. Frevert, T. W. Edminster and K. K. Barnes. 1966. Soil and Water Conservation Engineering. 2nd Ed. John Wiley & Sons, Inc. New York.

Shames, I. H. 1982. Mechanics of Fluids. McGraw-Hill Book Company. New York.

Swientek, R. J. 1986. On-line thickness control improves product quality. Food Processing. 47(8):100-101

Troeh, F. R., J. A. Hobbs and R. L. Donahue. 1980. Soil and Water Conservation for Productivity and Environment Protection. 2nd Ed. Prentice-Hall, Inc. New Jersey.

Vanoni, V. A., 1946. Transportation of suspended sediment by water. Transactions of the ASCE. 111:67-133.

Vanoni, V. A. and N. H. Brooks. 1957. Laboratory studies of the roughness and suspended load of alluvial streams. Final Report to Corps of Engineers, U. S. Army. Report No. E-68. Pasadena, California.

Wilson, B. N. and B. J. Barfield. 1986. Predicted and observed turbulence in detention ponds. Transactions of the ASAE. 29(5):1300-1306, 1313.

Wilson, B. N. and C. T. Rice. 1987. A large-scale laboratory apparatus for erosion research. ASAE Paper No. 87-2096, ASAE St. Joseph, MI 49085.

Wischmeier, W. H. 1955. Punch cards record runoff and soil loss data. Agricultural Engineering. 36:664-666.

Wischmeier, W. H., D. D. Smith and R. E. Uhland. 1958. Evaluation of factors in the soil loss equation. Agricultural Engineering. 39(8):458-462, 474.

Young, R. A. and C. K. Mutchler. 1969. Soil and water movement in small tillage channels. Transactions of the ASAE. 12(4):543-545.

APPENDIX A

CALIBRATION PROGRAMS

```
/* BLineCal.c
 * Base Line Calibration
*/

/* Standard Microsoft header files. */
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <conio.h>
#include <ctype.h>
#include <io.h>

/* Other header files. */
#include <iris.h>
#include <stddefs.h>
#include <baseline.h>
#include <2dSpace.h>

/* Define macros for vidio sync source. */
#define EXT_SYNC 1
#define INT_SYNC 0

/* Undefine the toupper macro so that the library function is used. */
#undef toupper

/* Declarations of external functions called from this file. */
extern      struct Line2D FindLine(int, int, int, int) ;
extern      WriteBaseLine(struct BaseLine *) ;
extern      ReadBaseLine(struct BaseLine *) ;
extern      double CalcZ0(double, double, double) ;
extern      void InitCondition(void) ;
extern      void Condition(void) ;

struct BaseLine BaseLine ;      /* This is the BaseLine structure which
                                   is to be filled then saved to disk. */

extern double SetPoint ;

main()
{
        char c ;
        int i, j, k, q ;

        printf("\n") ;
        printf("        **************************************************\n") ;
        printf("        *  B a s e   L i n e   C a l i b r a t i o n  *\n") ;
        printf("        **************************************************\n") ;
        printf("\n") ;

        is_initialize() ;
        is_reset() ;
        InitCondition() ;
        MakeBaseLine(&BaseLine) ;

        if(ERROR == WriteBaseLine(&BaseLine))
                puts("**** error writing baseline ***") ;
        if(ERROR == ReadBaseLine(&BaseLine))
                puts("**** error reading baseline ***") ;

        is_end() ;
}

/* MakeBaseLine()
 * Fill a BaseLine structure.
*/
MakeBaseLine(BaseLinePtr)
struct BaseLine *BaseLinePtr ;  /* Pointer to a BaseLine structure */
{
        int     n,              /* loop counter                         */
                temp,           /* temporary storage                    */
                XStart = 40,    /* minimum x value in window            */
                XEnd = 480,     /* maximum x value in window            */
                YStart = 100,   /* minimum y value in window            */
                YEnd = 400 ;    /* maximum y value in window            */

        float   CalBlock1Height,/* height of calibration block #1       */
                CalBlock2Height ;/* height of calibration block #2      */

        struct Line2D           /* (see 2dspace.h for Line2D structure) */
                Line0,          /* structure describing zero reference line. */
                Line1,          /* structure describing cal. block #1 line. */
                Line2 ;         /* structure describing cal. block #2 line. */

        unsigned
                MidY ;          /* y value of the midpoint of each line */

        double  MidX0,          /* x value of the midpoint of zero ref. line */
                MidX1,          /* x value of the midpoint of block #1 line. */
                MidX2,          /* x value of the midpoint of block #2 line. */
                AvgSlope,       /* average slope of the three lines.    */
                NewIntercept0,  /* new x intercept using average slope. */
                NewIntercept1,  /* new x intercept using average slope. */
                NewIntercept2 ; /* new x intercept using average slope. */

        /* Initialize beginning and end members of Base Line structure. */
        BaseLinePtr->Begin = YStart ;
        BaseLinePtr->End = YEnd ;

        /* Initialize ILUT #6 for thresholding at 50 */
        for(n = 0; n < 256; n++)
        {
                if(n < 50)
                        is_load_ilut_sval(6, n, 0) ;
                else
                        is_load_ilut_sval(6, n, n) ;
        }
```

```
/* Get line #0 (zero reference line) */
puts("\nPosition camera to view base line") ;
puts("    ... press any key to continue.") ;
Live(0) ;
getch() ;
is_freeze_frame() ;
Condition() ;
is_display(1) ;
Line0 = FindLine(XStart, XEnd, YStart, YEnd) ;

/* Get line #1 (calibration block #1) */
puts("\nPlace calibration block #1 in veiw") ;
puts("    ... press any key to continue.") ;
Live(0) ;
getch() ;
is_freeze_frame() ;
Condition() ;
is_display(1) ;
Line1 = FindLine(XStart, XEnd, YStart, YEnd) ;
puts("What is the calibration block's heigth?") ;
scanf(" %f", &CalBlock1Height) ;

/* Get line #2 (calibration block #2) */
puts("\nPlace calibration block #2 in veiw") ;
puts("    ... press any key to continue.") ;
Live(0) ;
getch() ;
is_freeze_frame() ;
Condition() ;
is_display(1) ;
Line2 = FindLine(XStart, XEnd, YStart, YEnd) ;
puts("What is the calibration block's heigth?") ;
scanf(" %f", &CalBlock2Height) ;

/* Calculate the average slope of the three lines */
AvgSlope = (Line0.Slope + Line1.Slope + Line2.Slope) / 3.0 ;

/* Calculate the X value at the midpoint of each line */
MidY = (BaseLinePtr->Begin + BaseLinePtr->End) / 2 ;
MidX0 = (Line0.Slope * MidY) + Line0.xIntercept ;
MidX1 = (Line1.Slope * MidY) + Line1.xIntercept ;
MidX2 = (Line2.Slope * MidY) + Line2.xIntercept ;

/* Using the X and Y values at the midpoint, calculate the
 * new intercept.
*/
NewIntercept0 = MidX0 - (MidY * AvgSlope) ;
NewIntercept1 = MidX1 - (MidY * AvgSlope) ;
NewIntercept2 = MidX2 - (MidY * AvgSlope) ;

/* Print some diagnostics. */
printf("Slopes: (0)%.4f   (1)%.4f   (2)%.4f   (avg)%.4f\n",
        Line0.Slope, Line1.Slope, Line2.Slope, AvgSlope) ;
        /* For each image row, calculate Xref, Zref, and K */
        for(n = YStart; n <= YEnd; n++)
        {
                double  DeltaX1,         /* pixels from line #0 to line #1 */
                        DeltaX2,         /* pixels from line #0 to line #2 */
                        V ;              /* intermediate variable (see
                                                derivation of equation) */

                /* Calculate difference between baseline and lines 1 & 2 */
                DeltaX1 = NewIntercept1 - NewIntercept0 ;       /* pixels */
                DeltaX2 = NewIntercept2 - NewIntercept0 ;       /* pixels */

                /* Calculate Xref in pixels */
                BaseLinePtr->Xref[n] = (AvgSlope * n) + NewIntercept0 ;

                /* Calculate V (V has no units) */
                V = (DeltaX1 * CalBlock2Height) / ( DeltaX2 * CalBlock1Height) ;

                /* Calculate Zref (same units as the CalBlockHeights) */
                BaseLinePtr->Zref[n] =
                      ((V * CalBlock1Height) - CalBlock2Height) / (V - 1.0) ;

                /* Calculate K */
                BaseLinePtr->K[n] =
                        DeltaX1 /
                        ( (1.0 / (BaseLinePtr->Zref[n] - CalBlock1Height))
                                        - (1.0 / BaseLinePtr->Zref[n]) ) ;
        }

        /* Mark BaseLine structure as being loaded */
        BaseLinePtr->Loaded = TRUE ;
}
```

```
/* FitLine.c */
/* Find the best fit line for an array of points in 2 dimensional space. */

/* Standard Microsoft header files. */
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

/* Other header files. */
#include <2dspace.h>

/* FitLineToPoints()
 * Find the line that best fits the given points.
 * This algorythm assumes the error of each point from a straight line
 * to be in the X direction.
 * NOTE that this function returns a structure.
*/
struct Line2D FitLineToPoints(Points, NPoints)
struct Point2D Points[] ;        /* Array of 2 dimensional points         */
int NPoints ;                    /* Number of points in Points[]          */
{
        unsigned int
                i ;              /* loop counter                          */

        struct Line2D            /* (see 2dspace.h for Line2D structure) */
                Line ;           /* structure defining a line.            */

        double  x, y,            /* temp. storage for point coordinates. */
                SigmaX,          /* Sum of x's                            */
                SigmaXX,         /* Sum of (x squared)'s                  */
                SigmaY,          /* Sum of y's                            */
                SigmaYY,         /* Sum of (y squared)'s                  */
                SigmaXY ;        /* Sum of (x * y)'s                      */

        /* Clear sums to zero. */
        SigmaX = SigmaY = SigmaXX = SigmaYY = SigmaXY = 0 ;

        /* For the array of points,                                       */
        /*      sum x's, y's, x squared's, y square's, and x*y's.         */
        for(i = 0; i < NPoints; i++)
        {
                x = Points[i].X ;
                y = Points[i].Y ;
                SigmaX += x ;
                SigmaY += y ;
                SigmaXX += x * x ;
                SigmaYY += y * y ;
                SigmaXY += x * y ;
        }

        /* Calculate best fit slope. */
        Line.Slope = ( SigmaXY - (SigmaY * SigmaX / NPoints) ) /
                     ( SigmaYY - (SigmaY * SigmaY / NPoints) ) ;
        /* Calculate best fit x intercept. */
        Line.xIntercept = ( SigmaX - (Line.Slope * SigmaY) ) / NPoints ;

        /* Return the line structure. */
        return(Line) ;
}
```

```
/* RWBaseLn.c */
/* Read Base Line and Write Base Line functions. */

/* Standard Microsoft header files. */
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <io.h>

/* Other header files. */
#include <stddefs.h>
#include <baseline.h>              /* Includes the BLINEFILE macro. */

/* Define the maximum line length for the BaseLine text file. */
#define MAX__LENGTH 80

/* Write a BaseLine structure to disk. */
ERRCODE WriteBaseLine(BaseLinePtr)
struct BaseLine *BaseLinePtr ;           /* Pointer to a BaseLine structure. */
{
        int      n ;               /* Loop counter                          */
        FILE     *fp ;             /* Reference to file stream for writing.*/

        /* Open the BaseLine file for writing in default mode (text). */
        if(NULL == (fp = fopen(BLINEFILE, "w")))
        {
                puts("*** error - unable to open BaseLine output file ***") ;
                return(ERROR) ;
        }

        /* Write the beginning and ending row number of the BaseLine. */
        fprintf(fp, "Begin = %d\nEnd = %d\n",
                                  BaseLinePtr->Begin, BaseLinePtr->End) ;

        /* For each row, write Xref, Zref, and K to the BaseLine file. */
        for(n = BaseLinePtr->Begin; n <= BaseLinePtr->End; n++)
        {
                fprintf(fp, "[%d] %f %f %f\n",
                        n,
                        BaseLinePtr->Xref[n],
                        BaseLinePtr->Zref[n],
                        BaseLinePtr->K[n]) ;
        }
        fclose(fp) ;
        return(NOERROR) ;
}

/* Read a BaseLine structure from disk. */
ERRCODE ReadBaseLine(BaseLinePtr)
struct BaseLine *BaseLinePtr ;
{
        int      n ;               /* Loop counter                          */
        FILE     *fp ;             /* Reference to file stream for writing.*/
        /* Open the BaseLine file for reading in default mode (text). */
        if(NULL == (fp = fopen(BLINEFILE, "r")))
        {
                perror("ReadBaseLine()") ;
                return(ERROR) ;
        }

        /* Read the beginning and ending row number of the BaseLine. */
        if(1 != fscanf(fp, " Begin = %d", &(BaseLinePtr->Begin)))
        {
                perror("ReadBaseLine()") ;
                fclose(fp) ;
                BaseLinePtr->Loaded = FALSE ;
                return(ERROR) ;
        }

        if(1 != fscanf(fp, " End = %d", &(BaseLinePtr->End)))
        {
                perror("ReadBaseLine()") ;
                fclose(fp) ;
                BaseLinePtr->Loaded = FALSE ;
                return(ERROR) ;
        }

        /* For each row, read Xref, Zref, and K from the BaseLine file. */
        for(n = BaseLinePtr->Begin; n <= BaseLinePtr->End; n++)
        {
                if(3 != fscanf(fp, " [%*d] %lf %lf %lf",
                                &(BaseLinePtr->Xref[n]),
                                &(BaseLinePtr->Zref[n]),
                                &(BaseLinePtr->K[n]) ))
                {
                        perror("ReadBaseLine()") ;
                        fclose(fp) ;
                        BaseLinePtr->Loaded = FALSE ;
                        return(ERROR) ;
                }
        }

        fclose(fp) ;

        /* Mark BaseLine structure as being loaded */
        BaseLinePtr->Loaded = TRUE ;

        return(NOERROR) ;
}
```

```
/* ScaleCal.c
 * Calibrate the X and Y Scaling factors.
*/

#include <conio.h>
#include <stdio.h>
#include <iris.h>
#include <stddefs.h>
#include "draw.h"
#include "measure.h"
#include "2dSpace.h"

#define LEFTEDGE        100
#define RIGHTEDGE       400
#define TOPEDGE         100
#define BOTTOMEDGE      400

extern void main(void) ;

static int Window[] = { LEFTEDGE, RIGHTEDGE, TOPEDGE, BOTTOMEDGE } ;
static int XWindow[] = { LEFTEDGE, RIGHTEDGE, TOPEDGE + 80, BOTTOMEDGE - 80 } ;
static int YWindow[] = { LEFTEDGE + 120, RIGHTEDGE - 120, TOPEDGE, BOTTOMEDGE } ;

void main()
{
        is_initialize() ;
        is_reset() ;
        is_set_sync_source(1) ;
        is_select_input_frame(0) ;
        is_select_output_frame(0) ;
        is_display(1) ;
        is_set_foreground(127) ;
        if(NOERROR == ScaleCalibrate(0))
                printf("Calibration Successful.\n") ;
        else
                printf("Calibration Unsuccessful.\n") ;
        IS_END() ;
}

ERRCODE ScaleCalibrate(Frame)
int Frame ;
{
        struct Line2D LeftLine, RightLine, TopLine, BottomLine ;
        double AvgVerticalSlope, AvgHorizontalSlope ;
        double MidLeftLineY, MidRightLineY, MidLeftLineX, MidRightLineX ;
        double MidTopLineY, MidBottomLineY, MidTopLineX, MidBottomLineX ;
        double DeltaX, DeltaY ;
        float  XDistance, YDistance ;
        double XFactor, YFactor ;
        int    ColOfPixels[512], RowOfPixels[512], col, row ;
        struct Point2D Points1[512], Points2[512] ;
        int    CompPix ;
        int    i ;
        int    MinValue, MinCol, MinRow ;
```

```
is_load_mask(0) ;
DrawBox(Window[0], Window[1], Window[2], Window[3]) ;
DrawBox(XWindow[0], XWindow[1], XWindow[2], XWindow[3]) ;
DrawBox(YWindow[0], YWindow[1], YWindow[2], YWindow[3]) ;
is_load_mask(1) ;
is_select_olut(7) ;

is_passthru() ;
printf("Place parallel lines in image...\n") ;
printf("    ... press any key to continue, <cr> to exit.\n") ;
if('\r' == getch())
        return(NOERROR) ;

is_acquire(0, 1) ;

for(i = 0, row = XWindow[2] + 1; row <= XWindow[3] - 1; i++, row++)
{
        is_get_pixel(0, row, 0, 512, RowOfPixels) ;

        /* Left side box. */
        MinValue = 256 ;
        for(col = Window[0] + 1; col <= YWindow[0] -1; col++)
        {
                if(RowOfPixels[col] < MinValue)
                {
                        MinValue = RowOfPixels[col] ;
                        MinCol = col ;
                }
        }
        Points1[i].Y = (float)row ;
        Points1[i].X = (float)MinCol ;
        CompPix = RowOfPixels[MinCol] ^ 0x80 ;
        is_put_pixel(0, row, MinCol, 1, &CompPix) ;

        /* Right side box. */
        MinValue = 256 ;
        for(col = YWindow[1] + 1; col <= Window[1] - 1; col++)
        {
                if(RowOfPixels[col] < MinValue)
                {
                        MinValue = RowOfPixels[col] ;
                        MinCol = col ;
                }
        }
        Points2[i].Y = (float)row ;
        Points2[i].X = (float)MinCol ;
        CompPix = RowOfPixels[MinCol] ^ 0x80 ;
        is_put_pixel(0, row, MinCol, 1, &CompPix) ;

}
LeftLine = FitLineToPoints(Points1, i - 1) ;
RightLine = FitLineToPoints(Points2, i - 1) ;
```

```
for(i = 0, col = YWindow[0] + 1; col <= YWindow[1] - 1; i++, col++)
{
        /* Top side box. */
        for(row = Window[2] + 1; row <= XWindow[2] - 1; row++)
        {
                is_get_pixel(0, row, col, 1, &(ColOfPixels[row])) ;
        }

        MinValue = 256 ;
        for(row = Window[2] + 1; row <= XWindow[2] - 1; row++)
        {
                if(ColOfPixels[row] < MinValue)
                {
                        MinValue = ColOfPixels[row] ;
                        MinRow = row ;
                }
        }
        Points1[i].X = (float)MinRow ;
        Points1[i].Y = (float)col ;
        CompPix = RowOfPixels[MinRow] ^ 0x80 ;
        is_put_pixel(0, MinRow, col, 1, &CompPix) ;

        /* Bottom side box. */
        for(row = XWindow[3] + 1; row <= Window[3] - 1; row++)
        {
                is_get_pixel(0, row, col, 1, &(ColOfPixels[row])) ;
        }
        MinValue = 256 ;
        for(row = XWindow[3] + 1; row <= Window[3] - 1; row++)
        {
                if(ColOfPixels[row] < MinValue)
                {
                        MinValue = ColOfPixels[row] ;
                        MinRow = row ;
                }
        }
        Points2[i].X = (float)MinRow ;
        Points2[i].Y = (float)col ;
        CompPix = ColOfPixels[MinRow] ^ 0x80 ;
        is_put_pixel(0, MinRow, col, 1, &CompPix) ;

}
TopLine = FitLineToPoints(Points1, i - 1) ;
BottomLine = FitLineToPoints(Points2, i - 1) ;

AvgVerticalSlope = (LeftLine.Slope + RightLine.Slope) / 2 ;
AvgHorizontalSlope = (TopLine.Slope + BottomLine.Slope) / 2 ;

MidLeftLineY = MidRightLineY = XWindow[3] - XWindow[2] ;
MidLeftLineX = (LeftLine.Slope * MidLeftLineY) + LeftLine.xIntercept ;
MidRightLineX = (RightLine.Slope * MidRightLineY) +
                                                        RightLine.xIntercept ;
        MidTopLineX = MidBottomLineX = YWindow[0] - YWindow[1] ;
        MidTopLineY = (TopLine.Slope * MidTopLineX) + TopLine.xIntercept ;
        MidBottomLineY = (BottomLine.Slope * MidBottomLineX) +
                                                        BottomLine.xIntercept ;

        DeltaX = MidRightLineX - MidLeftLineX ;
        DeltaY = MidBottomLineY - MidTopLineY ;
        printf("Delta X = %.4f      Delta Y = %.4f\n", DeltaX, DeltaY) ;

        printf("What is the distance between the two VERTICAL lines?\n") ;
        scanf(" %f", &XDistance) ;
        printf("What is the distance between the two HORIZONTAL lines?\n") ;
        scanf(" %f", &YDistance) ;

        XFactor = XDistance / DeltaX ;          /* inches per pixel     */
        YFactor = YDistance / DeltaY ;          /* inches per pixel     */
        printf("X factor = %.6f       Y factor = %.6f\n", XFactor, YFactor) ;

        if(ERROR == WriteScalingFactors(XFactor, YFactor))
                return(ERROR) ;
        else
                return(NOERROR) ;
}
```

APPENDIX B

SOIL PROFILE MEASUREMENT PROGRAMS

```
/* Measure.c */

/* Standard Microsoft header files. */
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <conio.h>
#include <ctype.h>
#include <io.h>
#include <string.h>

/* Other header files. */
#include <stddefs.h>
#include <iris.h>
#include <pcmotion.h>
#include <measure.h>
#include <topog.h>
#include <udm.h>

/* Stepping motor ramp table file names. */
#define RAMPTABLE__A "C:\\PCMOTION\\RAMPA.DAT"
#define RAMPTABLE__B "C:\\PCMOTION\\RAMPB.DAT"
#define RAMPTABLE__C "C:\\PCMOTION\\RAMPC.DAT"

#define INITIAL__THRESHOLD 50
#define UDM__TIMEOUT       1000

extern int errno ;
extern void Condition(void) ;
extern void InitCondition(void) ;
extern double SetPoint ;

struct  BaseLine BaseLine ;      /* baseline structure */
float   ImageXCenter = 512 / 2 ;/* pixel position   */
float   ImageYCenter = 480 / 2 ;/* pixel position   */
double  XScale, YScale ;         /* x and y scaling factor, inches per pixel */
double  XStepSize = .0156 ;      /* inches per step */
double  YStepSize = .02 ;        /* inches per step */
double  ZStepSize ;              /* inches per step (not currently used) */
BOOL     DoStep ;
int     ilutBuf[511] ;          /* Input lookup table buffer (help from KAN) */
int     xStepsPerSnap,
        yStepsPerSnap ;

/* Do initialization and settup for call to Measure() */
void main(argc, argv)
int argc ;              /* argument count                              */
char *argv[] ;          /* argument vectors                           */
{
        int errcode ;

        switch(argc)
        {
        /* Form: Measure <filename>      */
        case 2:
                DoStep = FALSE ;
                break ;

        /* Form: Measure <filename> STEP */
        case 3:
                if(0 == strcmpi(argv[2], "STEP"))
                {
                        DoStep = TRUE ;
                        break ;
                }else{
                        DoStep = FALSE ;
                }

        /* All other forms invalid.        */
        default:
                printf("usage: Measure <filename> [STEP]\n") ;
                return ;
        }

        /* Initialize stepping motor drivers. */
        PCM_Init() ;
        if( 0 != (errcode =
                PCM_LoadRampTables(RAMPTABLE_A, RAMPTABLE_B, RAMPTABLE_C)))
        {
                printf("error loading ramp tables, error code %d.\n", errcode) ;
                return ;
        }

        /* Initialize image processing hardware. */
        is_initialize() ;
        is_reset() ;
        InitCondition() ;

        /* Initialize ultra-sonic distance measurement device. */
        UDM_Init(UDM_BASE) ;

        /* Read in BaseLine */
        if(ERROR == ReadBaseLine(&BaseLine))
                return ;

        /* Read in scaling factors */
        if(ERROR == ReadScalingFactors(&XScale, &YScale))
                return ;

        /* Do measurement. */
        if(ERROR == Measure(argv[1]))
                printf("Measure Unsuccessful\n", &(*argv[1])) ;
        else
                printf("Measure Successful\n") ;

        is_end() ;
}
```

```
Measure(filename)
char *filename ;
{
        int     n,              /* Loop counter, array index.           */
                errcode,        /* Temporary storage for some functions
                                        return code.                    */
                argument,       /*                                      */
                XStart,         /* Minimum frame column to look for line. */
                XEnd,           /* Maximum frame column to look for line. */
                YStart,         /* Minimum frame row to look for line.  */
                YEnd,           /* Maximum frame row to look for line.  */
                xsteps,         /* Loop counter.                        */
                ysteps,         /* Loop counter.                        */
                thresh = INITIAL__THRESHOLD ; /* Threshold for conditioning. */

        static double
                XPos[512],      /* X position relative to camera position. */
                YPos[512],      /* Y position relative to camera position. */
                ZPos[512],      /* Z position relative to camera position. */
                xposition,      /* X position of camera.                */
                yposition,      /* Y position of camera.                */
                zposition = 0.0;/* Z position of camera.                */

        TOPOG   *tp ;           /* Pointer to file descriptor for an
                                        encoded topographic data file.  */

        char    c[2],           /* 1 character + null byte command string. */
                command ;       /* 1 character command. (1st char of c[])  */

        if(NULL == (tp = open__topog__write(filename)))
                return(ERROR) ;

        /* Set x, y, and z scaling factor for encoding. */
        new__topog__xscale(tp, XScale) ;
        new__topog__yscale(tp, YScale) ;
        new__topog__zscale(tp, 0.001) ;

        /* Define search window for line. */
        XStart = 50 ;
        XEnd = 450 ;
        YStart = BaseLine.Begin ;
        YEnd = BaseLine.End ;

        Live(0) ;

        do
        {
                printf("\n\nCOMMANDS:\n") ;
                printf("    X n ... Moves the camera 'n' steps in the X direction.\n") ;
                printf("    Y n ... Moves the camera 'n' steps in the Y direction.\n") ;
                printf("    Z n ... Moves the camera 'n' steps in the Z direction.\n") ;
                printf("    M ..... Begin surface measurements.\n") ;
                printf("    Q ..... Quit.\n\n") ;
        while(!kbhit())
        {
                UDM_Start(UDM_BASE) ;
                printf("height = %.2f\r",
                        UDM_Poll(UDM_BASE, UDM_TIMEOUT)
                        * 1111. * .0000016) ;
        }
        printf("\ncommand:") ;
        scanf(" %s", c) ;

        switch(command = c[0])
        {
        case 'x':
        case 'X':
                scanf("%d", &argument) ;
                PCM_MoveR(argument, 0, 0) ;
                break ;
        case 'y':
        case 'Y':
                scanf("%d", &argument) ;
                PCM_MoveR(0, argument, 0) ;
                break ;
        case 'z':
        case 'Z':
                scanf("%d", &argument) ;
                PCM_MoveR(0, 0, argument) ;
                break ;
        case 'q':
        case 'Q':
                goto quit ;
        }
}while('m' != tolower(command)) ;

printf("enter the number of steps per measurement in the x direction\n ") ;
printf("          (1 or 2 is typical): ") ;
scanf(" %d", &xStepsPerSnap) ;

PCM_MoveR(0, -50, 0) ;
PCM_Flag2() ;
PCM_MoveR(0, 50, 0) ;
PCM_Flag2() ;
PCM_MoveR(-50, 0, 0) ;
PCM_Flag1() ;
PCM_MoveR(50, 0, 0) ;
PCM_Flag1() ;

/* For each step in the y direction, do ... */
for(ysteps = 0; (yposition = (ysteps * YStepSize)) < 12.1;)
{
        /* For each step in the x direction, do ... */
        for(xsteps = 0; (xposition = (xsteps * XStepSize)) < 12.1;)
        {
            /* Make the display live. */
            Live(0) ;
```

```
/* Check to see if a key has been pressed. */
if(kbhit())
{
    /* If so then take action. */
    switch(getch())
    {
    /* if key = <esc>,
            abort x steps, continue at next y step. */
    case '\x1b':
        goto NextY ;

    /* if key = <return>,
            abort x and y steps. */
    case '\r':
        goto quit ;
    }
}

/* Snap this frame */
if(0 != is__freeze__frame())
{
    printf("... retrying.\n") ;
    continue ;
}

/* If stepping is enabled, step now and do processing
    while the camera stablizes. */
if(TRUE == DoStep)
{
        PCM__Flag1() ;
        PCM__Flag2() ;
        PCM__MoveR(xStepsPerSnap, 0, 0) ;
        xsteps += xStepsPerSnap ;
}

/* Do conditioning of image. */
Condition() ;
is__display(1) ;

/* Make a measurement.
 * Data goes into XPos[], YPos[], & ZPos[]. */
MeasureLineDepth(0, &BaseLine, XPos, YPos, ZPos,
        XStart, XEnd, YStart, YEnd, XScale, YScale) ;

/* Write each data point offset by the current camera
 * position to the encoded topog. file. */
for(n = YStart; n <= YEnd; n++)
{
    if(100.0 != ZPos[n])
    {
        if(NOERROR != write__topog(tp,
            XPos[n] + xposition,
            YPos[n] + yposition,
            ZPos[n] + zposition))
```

```
                                {
                                        printf("error writing topographical data.\n") ;
                                        return(ERROR) ;
                                }
                        }
                }
                printf("x y z %.4f %.4f %.4f\n",
                    xposition, yposition, zposition) ;
            }

            /* If stepping is enabled ...
             * make another pass. */
NextY:      if(TRUE == DoStep)
            {
                PCM__Flag1() ;
                PCM__MoveR(-xsteps - 50, 120, 0) ;
                PCM__Flag1() ;
                PCM__MoveR(50, 0, 0) ;
                ysteps += 120 ;
                xsteps = 0 ;
                PCM__Flag1() ;
            }
        }

        /* Close the topographic data file. */
quit:   if(0 != (errcode = close__topog__write(tp)))
        {
                printf("error closing topographical data file '%d'\n", errcode) ;
                printf("...errno = %d\n", errno) ;
                perror("") ;
                return(ERROR) ;
        }
        return(NOERROR) ;
}
```

```
/* baseline.h */

#ifndef BASELINE__H
#define BASELINE__H

struct BaseLine
{
        char Loaded ;           /* Structure loaded flag (set by
                                        MakeBaseLine and ReadBaseLine)  */
        int Begin, End ;        /* Beginning and ending defined
                                        array members.                  */
        double Xref[512] ;      /* Baseline pixel locations             */
        double Zref[512] ;      /* Baseline Zero on Z axis (heigth)     */
        double K[512] ;         /* Calibration constant                 */
} ;

#define BLINEFILE       "C:\\Measure\\Calib\\Baseline.cal"

#endif
```

```
/* RWScales.c */

#include <stdio.h>
#include <stddefs.h>
#include <measure.h>

#define SCALEFILE "C:\\measure\\calib\\xyscales.cal"

ERRCODE WriteScalingFactors(XScale, YScale)
double XScale, YScale ;
{
        FILE *fp ;

        if(NULL == (fp = fopen(SCALEFILE, "w")))
        {
                puts("**** error - unable to open BaseLine output file ***") ;
                return(ERROR) ;
        }

        fprintf(fp, "X Scale = %f (inches/pixel)\n", XScale) ;
        fprintf(fp, "Y Scale = %f (inches/pixel)\n", YScale) ;

        fclose(fp) ;
        return(NOERROR) ;
}

ERRCODE ReadScalingFactors(XScale, YScale)
double *XScale, *YScale ;
{
        FILE *fp ;
        double tXScale, tYScale ;

        if(NULL == (fp = fopen(SCALEFILE, "r")))
        {
                puts("**** error - unable to open input file ***") ;
                return(ERROR) ;
        }

        if(1 != fscanf(fp, " X Scale = %F (inches/pixel)", &tXScale))
        {
                fclose(fp) ;
                return(ERROR) ;
        }else{
                *XScale = tXScale ;
        }

        if(1 != fscanf(fp, " Y Scale = %F (inches/pixel)", &tYScale))
        {
                fclose(fp) ;
                return(ERROR) ;
        }else{
                *YScale = tYScale ;
        }
        fclose(fp) ;
        return(NOERROR) ;
}
```

```
/* 2dSpace.h */

#ifndef TWODSPACE__H
#define TWODSPACE__H

struct Point2D
{
        float X, Y ;
} ;

struct Line2D
{
        double Slope ;
        double xIntercept ;
        double rSquared ;
} ;

extern struct Line2D FitLineToPoints(struct Point2D [], int) ;

#endif
```

```
/* Measure.h */

#ifndef MEASURE__H
#define MEASURE__H

#include <stddefs.h>
#include <baseline.h>
#include "2dSpace.h"

#define CalcZ0(Zref, DeltaX, K) ((Zref)/(1.0+((DeltaX)*((Zref) / (K)))))

extern ERRCODE          WriteBaseLine(struct BaseLine *) ;
extern ERRCODE          ReadBaseLine(struct BaseLine *) ;
extern ERRCODE          WriteScalingFactors(double, double) ;
extern ERRCODE          ReadScalingFactors(double *, double *) ;

extern ERRCODE          ScaleCalibrate(int) ;

extern struct Line2D  FindLine(int, int, int, int) ;
extern struct Line2D  FindVerticalLine(int, int, int, int) ;
extern struct Line2D  FindHorizontalLine(int, int, int, int) ;
extern double         LineCenter(int [], int, int) ;
extern void           Live(int) ;
extern void           MeasureLineDepth(int, struct BaseLine *,
                          double [], double [], double [],
                          int, int, int, int, double, double) ;

#endif
```

```
/* MeasureLineDepth() */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <iris.h>
#include <conio.h>
#include <ctype.h>
#include <io.h>
#include <measure.h>

static int WhitePix = 255 ;
static int BlackPix  = 0 ;
extern float ImageXCenter, ImageYCenter ;

/* MeasureLineDepth()
 * Measure the line depth in the window defined by:
 *       XStart, XEnd, YStart, YEnd.
 * The coordinates of each point on the laser line are calculated
 *       and adjusted for "perspective distortion" and stored in
 *       arrays X[], Y[], and Z[]. Note that there is one set of coordinates
 *       for each image frame line from YStart to YEnd and they are stored
 *       in the X[], Y[], and Z[] array elements YStart to YEnd.
*/
void MeasureLineDepth(Frame, BaseLinePtr, X, Y, Z,
                                  XStart, XEnd, YStart, YEnd, XScale, YScale)
int      Frame ;                  /* Frame buffer number to process.       */
struct BaseLine *BaseLinePtr ;   /* Pointer to the BaseLine structure.   */
double X[], Y[], Z[] ;
int XStart, XEnd, YStart, YEnd ;/* Window limits to look for laser.      */
double XScale, YScale ;          /* Scaling factors to convert pixels to
                                         position (See ScaleCal()).       */
{
        register int
                n ;               /* Loop counter, array index.            */

        int     LineArray[512],/* Buffer for image row.                    */
                 showx,           /* Column to show center of laser line. */
                 showgrey ;       /* Grey level to show center of line.    */

        double   Z0,              /* Calculated distance from laser line
                                                                to camera. */
                 xPrime,          /* Uncompensated x position of line.     */
                 yPrime,          /* Uncompensated y position of line.     */
                 xAdj,            /* X coordinate adjusted for distortion.*/
                 yAdj ;           /* Y coordinate adjusted for distortion.*/

        /* Draw box to show window of interest. */
        DrawBox(XStart - 1, XEnd + 1, YStart - 1, YEnd + 1) ;

        /* For each display row ... */
        for(n = YStart; n <= YEnd; n++)
        {
            /* Read a row of pixels into LineArray[]. */
            is_get_pixel(Frame, n, 0, XEnd + 1, LineArray) ;

            /* Find the center of the line for this row. */
            xPrime = LineCenter(LineArray, XStart, XEnd) ;
/*          printf("center of line @ %3.1f\n", xPrime) ;          */

            /* If the center cannot be found then flag this set of
             * coordinates an invalid. LineCenter return -1.0 if it
             * cannot find the center of the line. The set of coordinates
             * is marked invalid by setting the Z[] array element to 100.0.
            */
            if(-1.0 == xPrime)
            {
                X[n] = Y[n] = Z[n] = 100.0 ;
            }else{
                /* yPrime is just the row number converted to a double. */
                yPrime = (double)n ;

                /* Determine frame column and grey level to show in frame.*/
                showgrey = LineArray[showx = (int)xPrime] ^ 0x80 ;

                /* Invert pixel at line center. */
                is_put_pixel(0, n, showx, 1, &showgrey) ;

                /* Calculate distance from camera to laser line. */
                Z0 = CalcZ0(BaseLinePtr->Zref[n],
                          xPrime - BaseLinePtr->Xref[n], BaseLinePtr->K[n]) ;

                /* Adjust x and y for "perspective distortion". */
                xAdj = ((xPrime - ImageXCenter)
                              * Z0 / BaseLinePtr->Zref[n]) + ImageXCenter ;
                yAdj = ((yPrime - ImageYCenter)
                              * Z0 / BaseLinePtr->Zref[n]) + ImageYCenter ;

                /* Store coordinates in X[], Y[], and Z[] arrays. */
                Z[n] = BaseLinePtr->Zref[n] - Z0 ;
                X[n] = XScale * xAdj ;
                Y[n] = YScale * yAdj ;
            }
        }
}
```

```
/* FindLine */
/* Functions to find vertical and horizontal lines. */

/* Standard Microsoft header files. */
#include <process.h>
#include <malloc.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

/* Other header files */
#include <iris.h>
#include <stddefs.h>
#include <2dSpace.h>

/* Function prototypes */
struct Line2D FindLine(int, int, int, int) ;
struct Line2D FindVerticalLine(int, int, int, int) ;
struct Line2D FindHorizontalLine(int, int, int, int) ;
double        LineCenter(int [], int, int) ;

static int WhitePix = 255 ;
static int BlackPix = 0 ;

/* Find a best fit vertical line in the window defined by
 * XStart, XEnd, YStart, and YEnd.
*/
struct Line2D FindVerticalLine(XStart, XEnd, YStart, YEnd)
int XStart, XEnd, YStart, YEnd ;         /* Window limits. */
{
        register int
                n,              /* Loop counter                    */
                row ;           /* Vidio frame row number          */

        int     RowOfPixels[512] ; /* Array to hold one line of pixels. */

        struct Point2D
                Points[512] ;   /* Array of points on the vertical line. */

        struct Line2D           /* (see 2dspace.h for Line2D structure */
                Line ;          /* Structure describing best fit line
                                                through Points. */

        /* For each row from YStart to YEnd... */
        /*     ... fill array of points with x and y coordinates. */
        for(row = YStart, n = 0; row <= YEnd; row++, n++)
        {
                int     CenterPix ;     /* Column # of center of line.  */

                /* Get a row of pixels */
                is_get_pixel(0, row, 0, 480, RowOfPixels) ;
                /* Y coordinate = row number */
                Points[n].Y = (float)row ;

                /* X coordinate = pixel number of center of line. */
                Points[n].X = LineCenter(RowOfPixels, XStart, XEnd) ;

                /* CenterPix = contrasting grey scale to center pixel. */
                CenterPix = RowOfPixels[(int)Points[n].X] ^ 0x80 ;
                is__put__pixel(0, row, (int)Points[n].X, 1, &CenterPix) ;
        }

        /* Find vertical line that best fits the points */
        Line = FitLineToPoints(Points, n) ;

        /* Show points of best fit line on display. */
        for(row = YStart; row <= YEnd; row++)
        {
                is__put__pixel(0, row,
                        (int)(Line.xIntercept + (Line.Slope * row) + 0.5),
                        1, &WhitePix) ;
        }

        return(Line) ;
}

/* Find a best fit horizontal line in the window defined by
 * XStart, XEnd, YStart, and YEnd.
*/
struct Line2D FindHorizontalLine(XStart, XEnd, YStart, YEnd)
int XStart, XEnd, YStart, YEnd ;
{
        register int
                n,              /* Loop counter                 */
                row,            /* Vidio frame row number        */
                column ;        /* Vidio frame row number        */

        int     ColumnOfPixels[512] ;/* Array to hold one line of pixels. */

        struct Point2D
                Points[512] ;   /* Array of points on the vertical line. */

        struct Line2D           /* (see 2dspace.h for Line2D structure */
                Line ;          /* Structure describing best fit line
                                                through Points. */

        /* For each column from YStart to YEnd... */
        /*      ... fill array of points with x and y coordinates. */
        for(column = XStart, n = 0; column <= XEnd; column++, n++)
        {
                int     CenterPix ;     /* Column # of center of line.  */

                /* Get a column of pixels */
                for(row = YStart; row <= YEnd; row++)
                        is__get__pixel(0, row, column, 1, &ColumnOfPixels[row]) ;
                /* Y coordinate = column number */
                Points[n].Y = (float)column ;

                /* X coordinate = pixel number of center of line. */
                Points[n].X = LineCenter(ColumnOfPixels, YStart, YEnd) ;

                /* CenterPix = contrasting grey scale to center pixel. */
                CenterPix = ColumnOfPixels[(int)Points[n].X] ^ 0x80 ;
                is_put_pixel(0, (int)Points[n].X, column, 1, &CenterPix) ;
        }

        /* Find horizontal line that best fits the points */
        Line = FitLineToPoints(Points, n) ;

        /* Show points of best fit line on display. */
        for(column = XStart; column <= XEnd; column++)
        {
                is_put_pixel(0,
                        (int)(Line.xIntercept + (Line.Slope * row) + 0.5),
                        column, 1, &WhitePix) ;
        }

        return(Line) ;
}

/* LineCenter()
 * Given an array of points, find the center of the non black points. */
double LineCenter(Array, LowerIndex, UpperIndex)
int Array[] ;           /* Array of grey scales.                        */
int LowerIndex ;         /* Lower array index limit for search for center. */
int UpperIndex ;         /* Upper array index limit for search for center. */
{
        register int
                i, j ;          /* Loop counters, array indecies.       */

        int     k,              /* Array index                              */
                UpperShelf,     /* Lower bounds of non black pixels.    */
                LowerShelf,     /* Upper bounds of non black pixels.    */
                ValidLowerShelf = FALSE, /* Flag indicating that the lower
                                shelf has been found without errors. */
                ValidUpperShelf = FALSE ;/* Flag indicating that the upper
                                shelf has been found without errors. */

        static int
                InitialGuess = -1  ; /* Starting index for initial search.
                                (-1 indicates not set yet) */
        if(-1 == InitialGuess)
                InitialGuess = (LowerIndex + UpperIndex) / 2 ;

        /* Look for white line, starting at InitialGuess */
        /*    Exit loop with k = index of a point on the line. */
        for(k = -1, i = j = InitialGuess;
                        (i > LowerIndex) && (j < UpperIndex); i--, j++)
        {
```

```
        if(i > LowerIndex)
        {
            if(Array[i] > 0)
            {
                k = i ;
                break ;
            }
        }
        if(j < UpperIndex)
        {
            if(Array[j] > 0)
            {
                k = j ;
                break ;
            }
        }
}

if(k < 0)
{
        InitialGuess = 256 ;
        return(-1.0) ;  /* error return */
}

/* LowerShelf = index of first dark pixel below Array[k] */
for(i = k; i > LowerIndex; i--)
{
        if(Array[i] == 0)
        {
            ValidLowerShelf = TRUE ;
            LowerShelf = i ;
            break ;
        }
}

/* UpperShelf = index of first dark pixel above Array[k] */
for(i = k; i < UpperIndex; i++)
{
        if(Array[i] == 0)
        {
            ValidUpperShelf = TRUE ;
            UpperShelf = i ;
            break ;
        }
}

/* If both shelves valid, return midpoint,
 *      else return 'not found' flag */
if(ValidUpperShelf && ValidLowerShelf)
{
        InitialGuess = (UpperShelf + LowerShelf) / 2 ;
        return(((double)UpperShelf + (double)LowerShelf) / 2.0) ;
}else{
        InitialGuess = 256 ;
```

```
                return(-1.0) ;  /* error return */
        }
}

/* Find a best fit vertical line in the window defined by
 * XStart, XEnd, YStart, and YEnd.
 * (same as FindVerticalLine but included for compatability with earlier
 * versions of some software. */
struct Line2D FindLine(XStart, XEnd, YStart, YEnd)
int XStart, XEnd, YStart, YEnd ;
{
        return(FindVerticalLine(XStart, XEnd, YStart, YEnd)) ;
}
```

APPENDIX C

VELOCITY MEASUREMENT PROGRAMS

```
/*Water.c
*/


#include <stdio.h>
#include <math.h>
#include <string.h>
#include <iris.h>
#include <conio.h>
#include <v2tov3.h>
#include <malloc.h>
#include "BlobScan.h"


#define MAXBLOB          100
#define BLACK            0
#define WHITE            1
#define MAXWHITE          255
#define FRAME__ZERO       0
#define FRAME__ONE        1
#define MIN__BLOB__AREA   20


struct Filter
{
        int rows ;
        int cols ;
        int divisor ;
        int *coefs ;
} ;


int LowPassCoefs[] = { 1, 1, 1, 1 } ;


struct Filter LowPass = { 2, 2, 4, LowPassCoefs } ;


struct BV__CalInfo
{
        int threshold ;
        int height, width, area ;
        float h__resolution, v__resolution ;
} ;


float xcenter[MAXBLOB][2], ycenter[MAXBLOB][2] ;
int Boarder[2][4] ;
int Frame = 1 ;
int top = 30 ;
int bottom = 450 ;
int left = 30 ;
int right = 480 ;
char calinfo__filename[] = "BV__Info.cal" ;
```

```
extern void FrameAdvance(int);


void main(void) ;
void tinue(void) ;
void moveit(void) ;
void acquire(int) ;
void convolve(int, int, struct Filter *) ;
void thresh(char *, int, int, int) ;
int  MoveThreshold(void) ;
void acq__ave(void) ;
void fastwtr(int, float, float) ;
struct BV__CalInfo *calibrate(struct BV__CalInfo *) ;
void slo__wtr(struct BV__CalInfo *) ;
static void DrawBox(int, int, int, int, int, int) ;
void SkipFrames(int) ;
void WaitForTrigger(void) ;
void DrawArc(int, int, int, int, int, int, int) ;


void main()
{
        int i, j ;
        char c;
        int m,n,d,filt[200];
        int blobthr;
        float res,wide;
        struct BV__CalInfo calinfo ;
        BOOL FirstTime ;
        FILE *file ;


        if(NULL == (file = fopen(calinfo__filename, "rb")))
        {
                FirstTime = TRUE ;
        }else{
                FirstTime = FALSE ;
                fread(&calinfo, sizeof(calinfo), 1, file) ;
                fclose(file) ;
        }


        is__initialize() ;
        is__allocate(2) ;
        is__select__ilut(0);
        is__select__input__frame(0);
        is__display(1);
        filt[0] = 0;


        for(;;)
        {
                char c ;
                if(FirstTime)
                {
                        c = 'c' ;
                        FirstTime = FALSE ;
                }else{
                        printf(" [C]alibrate [S]low [F]ast [Q]uit ?") ;
                        c = getch() ;
                        printf("\n") ;
                }
                switch(c)
                {
                case 'c':
                case 'C':
                        calibrate(&calinfo) ;
                        break ;
                case 's':
                case 'S':
                        slo_wtr(&calinfo) ;
                        break ;
                case 'f':
                case 'F':
/*                        fastwtr(&calinfo); */
                        break ;
                case 'q':
                case 'Q':
                        goto CleanUp ;
                }
        }
CleanUp:
        is_deallocate(2) ;
        is_end();
}


/*
 * calibrate()
*/
struct BV_CalInfo *calibrate(calinfo)
struct BV_CalInfo *calinfo ;
{
        char c = '\0';
        int b,i,j,n ;
        float ave,width;
        struct Blob *blob ;
        struct List *bloblist ;
        FILE *file ;


        is_cursor(0) ;
        is_select_input_frame(0) ;
        is_select_output_frame(0) ;
        is_select_ilut(0) ;
        is_select_olut(0) ;
```

```
puts("Enter the VERTICAL pixel resolution in inches/pixel") ;
scanf(" %f", &(calinfo->v_resolution)) ;
puts("Enter the HORIZONTAL pixel resolution in inches/pixel") ;
scanf(" %f", &(calinfo->h_resolution)) ;
puts("Play tape of beads traveling at velocity to be measured");
acquire(1) ;


convolve(FRAME_ZERO, FRAME_ONE, &LowPass) ;
is_select_output_frame(FRAME_ONE) ;


calinfo->threshold = MoveThreshold() ;
thresh("i", 1, calinfo->threshold, 255) ;
is_perform_feedback(FRAME_ONE, 1) ;
is_select_output_frame(FRAME_ONE) ;


is_select_ilut(0) ;
is_select_olut(0) ;


puts("scanning image...") ;
DrawBox(FRAME_ONE, BLACK, top, bottom, left, right) ;
if(NULL == (bloblist =
                Scan(FRAME_ONE, WHITE, top, bottom, left, right)))
{
        printf("Error scanning frame.\n") ;
        return ;
}


n = 0 ;
ave = 0 ;
for(blob = (struct Blob *)bloblist->Head;
                blob->node.Succ; blob = (struct Blob *)blob->node.Succ)
{
        if(blob->area > 20.0)
        {
                is_set_cursor_position((int)(blob->ycenter + .5),
                                        (int)(blob->xcenter + .5)) ;
                is_cursor(1) ;
                puts("Include this blob in width average? (y/n)");
                if('y' == getch())
                {
                        width = (blob->maxcol - blob->mincol) ;
                        ave += width ;
                        n++ ;
                        printf("Width = bead diameter = %f pixels\n",
                                        width) ;
                }
        }
}
```

```
        if(0 != n)
        {
                calinfo->width = ave / (float) n;
                printf("Average bead diameter = %d pixels\n", calinfo->width) ;
        }


        is_cursor(0);


        if(NULL == (file = fopen(calinfo_filename, "wb")))
        {
                puts("calibrate(): unable to open cal. info. file") ;
        }else{
                fwrite(calinfo, sizeof(*calinfo), 1, file) ;
                fclose(file) ;
        }


        return(calinfo) ;
}


void slo_wtr(calinfo)
struct BV_CalInfo *calinfo ;
{
        int c ;


        struct BlobMatch
        {
                struct Node node ;
                struct Blob *beforeblob, *afterblob ;
        } ;


        int b, num,numframe,i,j,num2 = 0;
        float vel, v2 = 0;
        struct Blob *blob1, *blob2 ;
        struct List *bloblist1, *bloblist2 ;
        struct BlobMatch *match ;
        struct List MatchList ;
        int Line[2] ;
        int NumMatches ;
        float SumXDistance, SumYDistance ;
        float XDistance, YDistance ;
        float SumXDistance_inches, SumYDistance_inches ;
        float XDistance_inches, YDistance_inches ;
        float dt = 3.0 / 30.0 ;


        InitList(&MatchList, NULL) ;
        is_set_sync_source(1) ;
        is_select_ilut(0) ;
```

```
is_select_olut(0) ;


puts("Locate FIRST frame of series to process and PAUSE VCR");
is_select_input_frame(0) ;
is_select_output_frame(0) ;
is_passthru() ;                     /* image into buf 0 */
WaitForTrigger() ;
is_freeze_frame() ;
is_select_input_frame(1) ;
is_select_output_frame(1) ;
is_passthru() ;
SkipFrames(3) ;
is_freeze_frame() ;
is_frame_copy(1, 2) ;
is_set_sync_source(0) ;


convolve(FRAME_ZERO, FRAME_ONE, &LowPass) ;
is_select_output_frame(FRAME_ONE) ;


thresh("i", 1, calinfo->threshold, 250) ;
is_perform_feedback(FRAME_ONE, 1) ;
is_select_output_frame(FRAME_ONE) ;


puts("scanning image...") ;
DrawBox(FRAME_ONE, BLACK, top, bottom, left, right) ;
if(NULL == (bloblist1 =
                Scan(FRAME_ONE, WHITE, top, bottom, left, right)))
{
        printf("Error scanning frame.\n") ;
        return ;
}


is_select_input_frame(0) ;
is_select_output_frame(0) ;
is_select_ilut(0) ;
is_select_olut(0) ;


is_frame_copy(2, 0) ;
is_select_output_frame(FRAME_ZERO) ;


convolve(FRAME_ZERO, FRAME_ONE, &LowPass) ;
thresh("i", 1, calinfo->threshold, 250) ;
is_perform_feedback(FRAME_ONE, 1) ;
is_select_output_frame(FRAME_ONE) ;
```

```
puts("scanning image...") ;
DrawBox(FRAME__ONE, BLACK, top, bottom, left, right) ;
if(NULL == (bloblist2 =
                Scan(FRAME__ONE, WHITE, top, bottom, left, right)))
{
        printf("Error scanning frame.\n") ;
        return ;
}


blob1 = (struct Blob *)bloblist1->Head ;
while(blob1->node.Succ)
{
        if(MIN__BLOB__AREA <= blob1->area)
        {
                DrawArc(FRAME__ONE, (int)(blob1->ycenter + 0.5),
                                (int)(blob1->xcenter + 0.5),
                                (int)(blob1->ycenter + 10.5),
                                (int)(blob1->xcenter + 0.5),
                                360, MAXWHITE) ;
        }
         blob1 = (struct Blob *)blob1->node.Succ ;
}


is__cursor(1) ;
blob1 = (struct Blob *)bloblist1->Head ;
blob2 = (struct Blob *)bloblist2->Head ;
while((bloblist1->Head->Succ != NULL) &&
                                (bloblist2->Head->Succ != NULL))
{
        /* If blob1 points to list header skip to 1st node. */
        if(NULL == blob1->node.Succ)
                blob1 = (struct Blob *)bloblist1->Head ;


        /* If blob2 points to list header skip to 1st node. */
        if(NULL == blob2->node.Succ)
                blob2 = (struct Blob *)bloblist2->Head ;


        /* Skip frame 1 blobs that are too small to be beads. */
        if(MIN__BLOB__AREA > blob1->area)
        {
                Remove(&(blob1->node)) ;
                blob1 = (struct Blob *)blob1->node.Succ ;
                continue ;
        }


        /* Skip frame 2 blobs that are too small to be beads. */
        if(MIN__BLOB__AREA > blob2->area)
        {
                Remove(&(blob2->node)) ;
        blob2 = (struct Blob *)blob2->node.Succ ;
        continue ;
}


printf("blob #%d at (row,col) (%3.1f,%3.1f)\n",
                blob1->number, blob1->ycenter, blob1->xcenter) ;


printf("\tblob #%d at (row,col) (%3.1f,%3.1f)\n",
                blob2->number, blob2->ycenter, blob2->xcenter) ;


DrawArc(FRAME__ONE, (int)(blob1->ycenter + 0.5),
                    (int)(blob1->xcenter + 0.5),
                    (int)(blob1->ycenter + 5.5),
                    (int)(blob1->xcenter + 0.5),
                    360, MAXWHITE) ;


is__set__cursor__position((int)(blob2->ycenter + 0.5),
                          (int)(blob2->xcenter + 0.5)) ;
printf("Match these two blobs?\n") ;
switch(getch())
{
case 'y':
        is__set__graphic__position((int)(blob1->ycenter + 0.5),
                                   (int)(blob1->xcenter + 0.5)) ;
        Line[0] = blob2->ycenter + 0.5 ;
        Line[1] = blob2->xcenter + 0.5 ;
        is__draw__lines(FRAME__ONE, 1, Line) ;


        if(NULL == (match = (struct BlobMatch *)
                        malloc(sizeof(struct BlobMatch))))
        {
                printf("unable to allocate match.\n") ;
                exit(1) ;
        }


        match->beforeblob =
                (struct Blob *)Remove(&(blob1->node)) ;
        match->afterblob =
                (struct Blob *)Remove(&(blob2->node)) ;
        Insert(&MatchList, &(match->node), NULL) ;
        blob1 = (struct Blob *)blob1->node.Succ ;
        blob2 = (struct Blob *)blob2->node.Succ ;
        break ;
case 'c':
        Remove(&(blob1->node)) ;
        DrawArc(FRAME__ONE, (int)(blob1->ycenter + 0.5),
                            (int)(blob1->xcenter + 0.5),
                            (int)(blob1->ycenter + 10.5),
                                        (int)(blob1->xcenter + 0.5),
                                        360, BLACK) ;
                        DrawArc(FRAME__ONE, (int)(blob1->ycenter + 0.5),
                                        (int)(blob1->xcenter + 0.5),
                                        (int)(blob1->ycenter + 5.5),
                                        (int)(blob1->xcenter + 0.5),
                                        360, BLACK) ;
                        blob1 = (struct Blob *)blob1->node.Succ ;
                        break ;
                case 'b':
                        Remove(&(blob2->node)) ;
                        blob2 = (struct Blob *)blob2->node.Succ ;
                        break ;
                default:
                        blob2 = (struct Blob *)blob2->node.Succ ;
                }
        }


        is__cursor(0);


        NumMatches = 0 ;
        SumXDistance = 0 ;
        SumYDistance = 0 ;
        SumXDistance__inches = 0.0 ;
        SumYDistance__inches = 0.0 ;
        for(match = (struct BlobMatch *)MatchList.Head;
                match->node.Succ; match = (struct BlobMatch *)match->node.Succ)
        {
                float dx, dy, dist ;
                float dx__inches, dy__inches, dist__inches ;


                SumXDistance += dx = (match->afterblob->xcenter -
                                                match->beforeblob->xcenter) ;
                SumYDistance += dy = (match->afterblob->ycenter -
                                                match->beforeblob->ycenter) ;
                dist = sqrt((dx * dx) + (dy * dy)) ;
                SumXDistance__inches += dx__inches = dx * calinfo->h__resolution ;
                SumYDistance__inches += dy__inches = dy * calinfo->v__resolution ;
                dist__inches = sqrt(
                    (dx__inches * dx__inches) + (dy__inches * dy__inches)) ;


                printf("blob #%d to blob #%d = x[%.2f] y[%.2f] dist[%.2f]\n",
                        match->beforeblob->number,
                        match->afterblob->number,
                        dx, dy, dist) ;
                printf("\tdistance = x[%.2f] y[%.2f] dist[%.2f] inches.\n",
                        dx__inches, dy__inches, dist__inches) ;
                printf("\tVELOCITY = x[%.2f] y[%.2f] vel[%.2f]\n\n",
                        dx__inches / dt / 12.0,
                        dy__inches / dt / 12.0,
                        dist__inches / dt / 12.0) ;


                NumMatches++ ;
        }


        if(NumMatches != 0)
        {
                XDistance = SumXDistance / (float)NumMatches ;
                YDistance = SumYDistance / (float)NumMatches ;
                XDistance__inches = SumXDistance__inches / (float)NumMatches ;
                YDistance__inches = SumYDistance__inches / (float)NumMatches ;
        }else{
                XDistance = YDistance = 0.0 ;
                XDistance__inches = YDistance__inches = 0.0 ;
        }


        printf("Average distance x[%.1f] y[%.1f] dist[%.1f]\n",
                        XDistance, YDistance,
                      sqrt((XDistance * XDistance) + (YDistance * YDistance))) ;


        printf("AVERAGE VELOCITY = x[%.2f] y[%.2f] vel[%.2f]\n\n",
                        XDistance__inches  / dt / 12.0,
                        YDistance__inches / dt / 12.0,
              sqrt((XDistance__inches * XDistance__inches) +
                   (YDistance__inches * YDistance__inches)) / dt / 12.0) ;
}


void tinue()
{
        char c;
        puts("Press any key to continue");
        scanf(" %c",&c);
}


void moveit()
{
        int frm;
        puts("Enter: 0 to move buf 1 to buf 0");
        puts("       1 to move buf 0 to buf 1");
        scanf("%d",&frm);
        if( frm ) is__copy__region(0,1,0,0);
        else is__copy__region(1,0,0,0);
}


void acquire(key)
int key;
{
        char chr;
        is_freeze_frame() ;
        is_frame_clear(FRAME_ZERO) ;
        DrawBox(FRAME_ZERO, MAXWHITE, top - 1, bottom + 1, left - 1, right + 1) ;
        is_set_sync_source(EXT_SYNC);
        is_passthru();
        if(key)
        {
                puts("Enter any key to acquire an image.") ;
                getch() ;
        }
        is_freeze_frame() ;
        is_set_sync_source(INT_SYNC) ;
}


void convolve(source, dest, filter)
int source, dest ;
struct Filter *filter ;
{
        if(filter->divisor == 0 )
        {
                puts("Filter has not been initialized. Retrieve or create.");
                return ;
        }else{
                printf("convolving image from buffer %d to buffer %d.\n",
                                        source, dest) ;
                is_convolve(source, dest, filter->rows, filter->cols,
                                                filter->coefs, filter->divisor) ;
        }
}


void thresh(in_out,tbl,gr_lv,new)
char *in_out ;
int tbl, gr_lv, new ;
{
        int i, red[256], green[256], blue[256] ;


        if((gr_lv < 0) || (gr_lv > 255))
        {
                printf("thresh(): threshold level of %d is out of range!\n",
                                        gr_lv) ;
                exit(1) ;
        }


        for(i = 0; i < gr_lv; ++i)
                red[i] = green[i] = blue[i] = 0 ;
        for(i = gr_lv; i < 256; ++i)
                red[i] = green[i] = blue[i] = new ;


        switch(in_out[0])
        {
        case 'i':
        case 'I':
                is_load_ilut(tbl, green) ;
                is_select_ilut(tbl) ;
                break ;
        case 'o':
        case 'O':
                is_load_olut( tbl,red,green,blue );
                is_select_olut(tbl);
                break ;
        default:
                printf("thresh(): invalid parameter 1.\n") ;
        }
}


int MoveThreshold()
{
        char c;
        static int threshold = 128 ;


        puts("Adjust threshold:[u]p, [U]p 10, [d]own, [D]own 10, <cr> to quit");
        thresh("o" , 1, threshold, 255) ;
        while( '\r' != (c = getch()))
        {
                switch(c)
                {
                case 'u':
                        threshold++ ;
                        break ;
                case 'd':
                        threshold-- ;
                        break ;
                case 'U':
                        threshold += 10 ;
                        break ;
                case 'D':
                        threshold -= 10 ;
                        break ;
                }
                threshold = max(0, min(255, threshold)) ;
                thresh("o" , 1, threshold, 255) ;
        }
        printf("Threshold chosen at grey level %d.\n", threshold);
        return(threshold) ;
}
```

```
void acq_ave()
{
        int numframes;


        is_set_sync_source(1) ;
        is_select_output_frame(0) ;
        is_passthru() ;
        puts("Key the number of frames to be averaged, <cr> to acquire.") ;
        scanf("%d",&numframes);
        while(numframes--)
                FrameAdvance(1);
        is_set_sync_source(0);
}


DrawCross(x, y)
int x, y ;
{
        int px, py, WhitePixel = 255 ;


        for(px = x - 5; px < x + 5; px++)
                is_put_pixel(0, y, px, 1, &WhitePixel) ;
        for(py = y - 5; py < y + 5; py++)
                is_put_pixel(0, py, x, 1, &WhitePixel) ;
}


static void DrawBox(frame, color, top, bottom, left, right)
int frame, color, top, bottom, left, right ;
{
        int array[8] ;


        array[0] = top ;
        array[1] = left ;
        array[2] = top ;
        array[3] = right ;
        array[4] = bottom ;
        array[5] = right ;
        array[6] = bottom ;
        array[7] = left ;
        is_set_foreground(color) ;
        is_set_graphic_position(bottom, left) ;
        is_draw_lines(frame, 4, array) ;
}


void SkipFrames(count)
int count ;
{
        unsigned csr ;
        count -= 2 ;
        while(count-- > 0)
        {
                do{
                        csr = inpw(OUTCSR) ;
                }while(!(((csr & 0x8000) != 0) && ((csr & 0x2000) == 0))) ;


                do{
                        csr = inpw(OUTCSR) ;
                }while(!(((csr & 0x8000) == 0) && ((csr & 0x2000) == 0))) ;
        }
}


void WaitForTrigger()
{
        getch() ;
}


void DrawArc(frame, cy, cx, y, x, angle, color)
int cx, cy, x, y, angle, color ;
{
        is_set_foreground(color) ;
        is_set_graphic_position(y, x) ;
        is_draw_arc(frame, cy, cx, angle) ;
}
```

```
/* BlobScan.c
 * Connectivity Analysis.
 *
 * Based on the article:
 *       Segmenting Binary Images
 *       Robotics Age, July/August 1981,
 *       Vol. 3, No. 4, pp. 4-19
 *       by Robert Cunningham
 *       Robitics Research Program
 *       NASA Jet Propulsion Laboratory
 *       California Institute of Technology
 *       Pasadena, California
*/


#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>
#include "BlobScan.h"


#define NextPrevRun(env) (env->PrevRun = (struct Run *)env->PrevRun->node.Pred)


/* Run-length structure. */
struct Run
{
        struct Node node ;
        struct VBlob *vblob ;
        int startcol ;
        int row ;
        int length ;
        int number ;
} ;


static struct VBlob *NewVBlob(struct BlobScanEnvironment *, int, int, int, int) ;
static struct Run    *NewRun(struct BlobScanEnvironment *,
                                        struct VBlob *, int, int, int) ;
static void           GetLine(int, int [], int) ;
static int            GetPixel(int, int, int) ;
static void           Update(struct BlobScanEnvironment *, BOOL) ;
static struct VBlob *LookUp(struct Run *) ;
static void           state1(struct BlobScanEnvironment *) ;
static void           state2(struct BlobScanEnvironment *) ;
static void           state3(struct BlobScanEnvironment *) ;
static void           state4(struct BlobScanEnvironment *) ;
static void           state6(struct BlobScanEnvironment *) ;
static void           state7(struct BlobScanEnvironment *) ;
static void           Debug(struct BlobScanEnvironment *) ;
static void           FreeNodes(struct List *) ;
```

```
BlobScan(env)
struct BlobScanEnvironment *env ;
{
        struct VBlob *vblob ;
        struct VBlob *BGvblob ;
        struct Run *run ;
        int blobnum ;


        /* Initialize the BlobScanEnvironment structure. */
        InitBlobScanEnvironment(env) ;


        /* Clear the previous line array. */
        for(env->col = env->colstart; env->col <= env->colend; env->col++)
                env->PrevLine[env->col] = 0 ;


        /* Create and initialize background blob. */
        BGvblob = NewVBlob(env, GetPixel(env->frame, env->rowstart,
                    env->colstart), env->colstart, env->colend, env->rowstart) ;
        Insert(&(env->vbloblist), &(BGvblob->node), NULL) ;


        /* Create and initialize first and second run. */
        env->PrevRun =
                  NewRun(env, BGvblob, env->colstart,
                          env->rowstart, env->colend - env->colstart + 1) ;
        Insert(&(env->runlist), &(env->PrevRun->node), NULL) ;


        env->CurrentRun = NewRun(env, BGvblob, env->colstart,
                                                   env->rowstart + 1, 0) ;
        Insert(&(env->runlist), &(env->CurrentRun->node), NULL) ;


        for(env->row = env->rowstart + 1; env->row <= env->rowend; env->row++)
        {
                env->NewBlobFlag = FALSE ;
                env->state = 0 ;


                GetLine(env->frame, env->CurrentLine, env->row) ;


                /* Set CurrentBlob and AboveBlob to the Background Blob. */
                env->CurrentVBlob = env->AboveVBlob = LookUp(env->PrevRun) ;


                for(env->col = env->colstart + 1;
                                        env->col <= env->colend; env->col++)
                {
                        env->state = (env->state / 4) +
                                (4 * env->PrevLine[env->col]) +
                    (8 * env->CurrentLine[env->col]) ;
                env->PrevLine[env->col] = env->CurrentLine[env->col] ;
                switch(env->state)
                {
                case 7:
                case 8:
                        state7(env) ;
                        break ;
                case 4:
                case 11:
                        state4(env) ;
                        break ;
                case 3:
                case 12:
                        state3(env) ;
                        break ;
                case 2:
                case 13:
                        state2(env) ;
                        break ;
                case 6:
                case 9:
                        state6(env) ;
                        break ;
                case 1:
                case 14:
                        state1(env) ;
                        break ;
                case 0:
                case 15:
                case 5:
                case 10:
                        break ;
                default:
                        printf("illegal state: %d\n", env->state) ;
                        exit(1) ;
                }
        }
        Update(env, !NOEOL) ;
}


for(vblob = (struct VBlob *)env->vbloblist.TailPred, blobnum = 1;
            vblob->node.Pred; vblob = (struct VBlob *)vblob->node.Pred)
{
        if(vblob->blob->node.Type == ACTUALBLOB__TYPE)
        {
                Insert(&(env->bloblist), &(vblob->blob->node), NULL) ;
                vblob->blob->number = blobnum++ ;
                vblob->blob->xcenter = vblob->blob->sumcol /
                                                vblob->blob->area ;
                vblob->blob->ycenter = vblob->blob->sumrow /
                                                vblob->blob->area ;
        }else{
```

```
                        vblob->actualblob.number = 0;
                }
        }
}


static void state1(env)
struct BlobScanEnvironment *env ;
{
        struct VBlob *vblob, *vblob2 ;
        int i ;
        int  *int__ptr ;
        static char *fmt ;
        int here = 0 ;


        env->HoleVBlob = env->AboveVBlob ;
        NextPrevRun(env) ;


        /* AboveBlob will point to a VBlob containing an actual blob
         * throughout the remainder of the function. */
        env->AboveVBlob = LookUp(env->PrevRun) ;


        if(env->NewBlobFlag)
        {
                env->NewBlobFlag = FALSE ;
                env->CurrentVBlob = env->AboveVBlob ;
                env->CurrentVBlob->actualblob.perimeter +=
                                                     env->col - env->LeftEnd ;
        }else{
                if(env->CurrentVBlob == env->AboveVBlob)
                {
                        env->HoleVBlob->actualblob.sibling =
                                       env->CurrentVBlob->actualblob.child ;
                        env->HoleVBlob->actualblob.parent = env->CurrentVBlob ;
                        env->CurrentVBlob->actualblob.child =  env->HoleVBlob ;
                        (env->CurrentVBlob->actualblob.NumHoles)++ ;
                        env->CurrentVBlob->actualblob.perimeter -=
                                        env->HoleVBlob->actualblob.perimeter ;
                }else{
                        env->CurrentVBlob->actualblob.perimeter +=
                                env->AboveVBlob->actualblob.perimeter +
                                env->col - env->LeftEnd ;
                        if(env->AboveVBlob->actualblob.mincol <
                                       env->CurrentVBlob->actualblob.mincol)
                        {
                                 env->CurrentVBlob->actualblob.mincol =
                                       env->AboveVBlob->actualblob.mincol ;
                        }
                        if(env->AboveVBlob->actualblob.maxcol >
                                        env->CurrentVBlob->actualblob.maxcol)
                        {
                                 env->CurrentVBlob->actualblob.maxcol =
                                        env->AboveVBlob->actualblob.maxcol ;
                        }


                        if(env->AboveVBlob->actualblob.maxrow <
                                        env->CurrentVBlob->actualblob.maxrow)
                        {
                                env->CurrentVBlob->actualblob.maxrow =
                                        env->AboveVBlob->actualblob.maxrow ;
                        }


                        env->CurrentVBlob->actualblob.area +=
                                        env->AboveVBlob->actualblob.area ;
                        env->CurrentVBlob->actualblob.sumcol +=
                                        env->AboveVBlob->actualblob.sumcol ;
                        env->CurrentVBlob->actualblob.sumrow +=
                                        env->AboveVBlob->actualblob.sumrow ;
                        env->CurrentVBlob->actualblob.sumcol2 +=
                                        env->AboveVBlob->actualblob.sumcol2 ;
                        env->CurrentVBlob->actualblob.sumrowcol +=
                                        env->AboveVBlob->actualblob.sumrowcol ;
                        env->CurrentVBlob->actualblob.sumrow2 +=
                                        env->AboveVBlob->actualblob.sumrow2 ;


                        if(env->AboveVBlob->actualblob.NumHoles != 0)
                        {
                                env->CurrentVBlob->actualblob.NumHoles +=
                                        env->AboveVBlob->actualblob.NumHoles ;
                                vblob = env->AboveVBlob->actualblob.child ;
                                while(vblob != NULL)
                                {
                                        vblob->actualblob.parent =
                                              env->CurrentVBlob ;
                                        vblob2 = vblob ;
                                        vblob = vblob2->actualblob.sibling ;
                                }
                                vblob2->actualblob.sibling =
                                            env->CurrentVBlob->actualblob.child ;
                                env->CurrentVBlob->actualblob.child =
                                               env->AboveVBlob->actualblob.child ;
                        }
                        env->AboveVBlob->blob =
                                            (struct Blob *)env->CurrentVBlob ;
                        env->AboveVBlob = env->CurrentVBlob ;
                }
        }
        env->HoleVBlob->actualblob.perimeter += env->col - env->LeftEnd ;
}
```

```
static void state2(env)
struct BlobScanEnvironment *env ;
{
        struct VBlob *vblob ;


        if(env->NewBlobFlag)
        {
                env->CurrentVBlob = NewVBlob(env,
                                env->CurrentLine[env->CurrentRun->startcol],
                                env->CurrentRun->startcol, env->col, env->row) ;
                Insert(&(env->vbloblist), &(env->CurrentVBlob->node), NULL) ;
        }
        Update(env, NOEOL) ;
        env->CurrentVBlob->actualblob.perimeter += env->col - env->LeftEnd ;
        env->AboveVBlob->actualblob.perimeter += env->col - env->LeftEnd ;
        env->CurrentVBlob = env->AboveVBlob ;
}


static void state3(env)
struct BlobScanEnvironment *env ;
{
        Update(env, NOEOL) ;
        NextPrevRun(env) ;
        env->CurrentVBlob = env->AboveVBlob = LookUp(env->PrevRun) ;
}


static void state4(env)
struct BlobScanEnvironment *env ;
{
        env->LeftEnd = env->col ;
        NextPrevRun(env) ;
        env->AboveVBlob = LookUp(env->PrevRun) ;
}


static void state6(env)
struct BlobScanEnvironment *env ;
{
        struct VBlob *vblob ;


        if(env->NewBlobFlag)
        {
                env->CurrentVBlob = NewVBlob(env,
                               env->CurrentLine[env->CurrentRun->startcol],
                               env->CurrentRun->startcol, env->col, env->row) ;
                Insert(&(env->vbloblist), &(env->CurrentVBlob->node), NULL) ;
                env->NewBlobFlag = FALSE ;
        }
        Update(env, NOEOL) ;
        env->CurrentVBlob->actualblob.perimeter += env->col - env->LeftEnd ;
        env->AboveVBlob->actualblob.perimeter += env->col - env->LeftEnd ;
        env->CurrentVBlob = env->AboveVBlob ;
        NextPrevRun(env) ;
        env->AboveVBlob = LookUp(env->PrevRun) ;
        env->LeftEnd = env->col ;
}


static void state7(env)
struct BlobScanEnvironment *env ;
{
        Update(env, NOEOL) ;
        env->LeftEnd = env->col ;
        env->NewBlobFlag = TRUE ;
}


/* Determine which VBlob a Run belongs to. */
static struct VBlob *LookUp(run)
struct Run *run ;
{
        struct VBlob *vblob ;


        if(run->node.Type != RUN__TYPE)
        {
                printf("INVALID RUN FOR LookUp(): run #%d  type %d\n",
                                                run->number, run->node.Type) ;
                exit(1) ;
        }


        /* Chain through blob pointers until an actual blob is found. */
        for(vblob = run->vblob; vblob->blob->node.Type == VBLOB__TYPE;
                                        vblob = (struct VBlob *)vblob->blob)
                ;


        if(vblob->blob->node.Type != ACTUALBLOB__TYPE)
        {
                printf("ILLEGAL NODE TYPE FOR LookUp(): vblob #%d  blobtype %d\n",
                        vblob->number, vblob->blob->node.Type) ;
                exit(1) ;
        }


        /* Return pointer to vblob containing the actual blob. */
        return(vblob) ;
}


struct BlobScanEnvironment *InitBlobScanEnvironment(env)
struct BlobScanEnvironment *env ;
{
        InitList(&(env->vbloblist), VBLOBLIST__TYPE) ;
        InitList(&(env->runlist), RUNLIST__TYPE) ;
        InitList(&(env->bloblist), BLOBLIST__TYPE) ;
        env->NumBlobs = 0 ;
        env->NumVBlobs = 0 ;
        env->NumRuns = 0 ;
        return(env) ;
}


static struct VBlob *NewVBlob(env, color, colstart, colend, rowstart)
struct BlobScanEnvironment *env ;
int color, colstart, colend, rowstart ;
{
        struct VBlob *vblob ;
        struct Blob *ablob ;


        env->NewBlobFlag = FALSE ;
        if(NULL == (vblob = (struct VBlob *)malloc(sizeof(struct VBlob))))
        {
                perror("NewVBlob()") ;
                exit(1) ;
        }


        /* Initialize Virtual Blob. */
        vblob->node.Type = VBLOB__TYPE ;
        vblob->number = ++(env->NumVBlobs) ;
        ablob = vblob->blob = &(vblob->actualblob) ;


        /* Initialize Actual Blob. */
        ablob->node.Type = ACTUALBLOB__TYPE ;
        ablob->number = ++(env->NumBlobs) ;
        ablob->parent = NULL ;
        ablob->child = NULL ;
        ablob->sibling = NULL ;
        ablob->color = color ;
        ablob->perimeter = 0 ;
        ablob->NumHoles = 0 ;
        ablob->area = 0 ;
        ablob->sumrow = 0 ;
        ablob->sumcol = 0 ;
        ablob->sumrow2 = 0 ;
        ablob->sumcol2 = 0 ;
        ablob->sumrowcol = 0 ;
        ablob->mincol = colstart ;
        ablob->maxcol = colend ;
        ablob->minrow = rowstart ;
        ablob->maxrow = rowstart ;
        return(vblob) ;
}
```

```
static struct Run *NewRun(env, vblob, startcol, row, length)
struct BlobScanEnvironment *env ;
struct VBlob *vblob ;
int startcol ;
int row ;
int length ;
{
        struct Run *run ;


        if(NULL == (run = (struct Run *)malloc(sizeof(struct Run))))
        {
                perror("NewRun()") ;
                exit(1) ;
        }
        run->node.Type = RUN_TYPE ;
        run->node.Pred = NULL ;
        run->node.Succ = NULL ;
        run->vblob = vblob ;
        run->startcol = startcol ;
        run->row = row ;
        run->length = length ;
        run->number = ++(env->NumRuns) ;
        return(run) ;
}


static void GetLine(frame, buf, row)
int frame ;
int buf[] ;
int row ;
{
        int i ;


        is_get_pixel(frame, row, 0, MAXCOL, buf) ;
        for(i = 0; i < MAXCOL; i++)
        {
                if(buf[i] != 0)
                        buf[i] = 1 ;
        }
}


static int GetPixel(frame, row, col)
int frame ;
int row, col ;
{
        int pixel ;


        is_get_pixel(frame, row, col, 1, &pixel) ;
        if(pixel != 0)
                pixel = 1 ;
        return(pixel) ;
}


static void Update(env, EOLflag)
struct BlobScanEnvironment *env ;
BOOL EOLflag ;
{
        double length, start ;
        double frow = env->row ;


        length = env->CurrentRun->length =
                                        env->col - env->CurrentRun->startcol ;


        env->CurrentRun->vblob = env->CurrentVBlob ;


        env->CurrentVBlob->actualblob.perimeter += 2 ;


        start = env->CurrentRun->startcol ;


        env->CurrentVBlob->actualblob.area += length ;
        env->CurrentVBlob->actualblob.sumcol +=
                        length * ((length - 1) / 2 + start) ;
        env->CurrentVBlob->actualblob.sumrow +=
                        length * frow ;
        env->CurrentVBlob->actualblob.sumcol2 +=
                        length *
                        (       (length - 1) *
                                (start + (2 * length - 1) / 6) +
                                (start * start)
                        ) ;
        env->CurrentVBlob->actualblob.sumrowcol +=
                        length * ((length - 1) / 2 + start) * frow ;
        env->CurrentVBlob->actualblob.sumrow2 +=
                        length * frow * frow ;


        if(env->CurrentRun->startcol < env->CurrentVBlob->actualblob.mincol)
                        env->CurrentVBlob->actualblob.mincol =
                                env->CurrentRun->startcol ;


        if(env->col - 1 > env->CurrentVBlob->actualblob.maxcol)
                env->CurrentVBlob->actualblob.maxcol = env->col - 1 ;


        env->CurrentVBlob->actualblob.maxrow = env->row ;
        env->CurrentRun =
                      NewRun(env, env->CurrentVBlob, env->col, env->row, 0) ;
        Insert(&(env->runlist), &(env->CurrentRun->node), NULL) ;
        if(EOLflag != NOEOL)
        {
                env->CurrentRun->startcol = env->colstart ;
                NextPrevRun(env) ;
        }
}


static void Debug(env)
struct BlobScanEnvironment *env ;
{
        static int firsttime = TRUE ;
        static struct VBlob *Current ;
        char c ;
        int i ;


        if(firsttime)
        {
                Current = (struct VBlob *)&(env->vbloblist) ;
                firsttime = FALSE ;
        }


        is__set__cursor__position(env->row, env->col) ;


        for(;;)
        {
                struct VBlob *vblob ;
                struct Run *run ;


                printf("VBlobList:\n") ;
                for(vblob = (struct VBlob *)env->vbloblist.Head;
                     vblob->node.Succ; vblob = (struct VBlob *)vblob->node.Succ)
                {
                        printf("vblob #%d -> actualblob #%d", vblob->number,
                               (vblob->blob->node.Type == ACTUALBLOB__TYPE)?
                                 vblob->blob->number :
                                 ((struct VBlob *)(vblob->blob))->blob->number) ;
                        if(vblob == Current)
                        {
                                printf(" (current)\n") ;
                                if(vblob->actualblob.parent != NULL)
                                        printf("\tparent = vblob #%d  ",
                                               vblob->actualblob.parent->number) ;
                                else
                                        printf("\tparent = NONE  ") ;
                                if(vblob->actualblob.child != NULL)
                                        printf("\tchild = vblob #%d  ",
                                                vblob->actualblob.child->number) ;
                                else
                        printf("\tchild = NONE  ") ;
                if(vblob->actualblob.sibling != NULL)
                        printf("\tsibling = vblob #%d\n",
                             vblob->actualblob.sibling->number) ;
                else
                        printf("\tsibling = NONE\n") ;
                printf("\tcolor = %s\n",
                        (vblob->actualblob.color == 0) ?
                        "BLACK" : "WHITE") ;
                printf("\tcolumn: min = %d max = %d  ",
                        vblob->actualblob.mincol,
                        vblob->actualblob.maxcol) ;
                printf("\trow: min = %d max = %d\n",
                        vblob->actualblob.minrow,
                        vblob->actualblob.maxrow) ;
                printf("\tperimeter = %d  NumHoles = %d\n",
                        vblob->actualblob.perimeter,
                        vblob->actualblob.NumHoles) ;
                printf("\tarea = %f\n",
                        vblob->actualblob.area) ;
                printf("\tsumcol = %-20f  sumrow = %-20f\n",
                        vblob->actualblob.sumcol,
                        vblob->actualblob.sumrow) ;
                printf("\tsumcol2 = %-20f sumrow2 = %-20f\n",
                        vblob->actualblob.sumcol2,
                        vblob->actualblob.sumrow2) ;
                printf("\tsumrowcol = %-20f\n",
                        vblob->actualblob.sumrowcol) ;
        }else{
                printf(" \n") ;
        }
}


switch(c = getch())
{
case 's':
        if(Current->node.Succ != 0)
                Current = (struct VBlob *)
                                Current->node.Succ ;
        break ;
case 'p':
        if(Current->node.Pred != 0)
                Current = (struct VBlob *)
                                Current->node.Pred ;
        break ;
case '\x20':
        printf("\n\n") ;
        return ;
case 'q':
        exit(0) ;
default:
        printf("what?\n") ;
}
```

```
                printf("\n") ;
        }
}


struct List *Scan(frame, color, top, bottom, left, right)
int frame ;      /* Frame number to scan (0 or 1). */
int color ;      /* Color of blobs to include in returned list. */
int top, bottom,
    left, right ; /* Boarder positions. */
{
        struct BlobScanEnvironment env ;
        struct Blob *blob, *nextblob, *newblob ;
        struct List *bloblist ;


        env.frame = frame ;
        env.rowstart = top ;
        env.rowend = bottom ;
        env.colstart = left ;
        env.colend = right ;


        if(NULL == (bloblist = AllocList()))
                return(NULL) ;


        BlobScan(&env) ;


        for(blob = (struct Blob *)env.bloblist.Head;
                    nextblob = (struct Blob *)blob->node.Succ; blob = nextblob)
        {
                if(blob->color == color)
                {
                        if(NULL == (newblob = (struct Blob *)
                                                malloc(sizeof(struct Blob))))
                        {
                                puts("Scan():unable to allocate new blob.") ;
                                return(NULL) ;
                        }


                        *newblob = *blob ;


                        Insert(bloblist, &(newblob->node), NULL) ;
                }
        }


        CleanUpBlobScan(&env) ;
        return(bloblist) ;
}


void CleanUpBlobScan(env)
struct BlobScanEnvironment *env ;
{
        FreeNodes(&(env->vbloblist)) ;
        FreeNodes(&(env->runlist)) ;
}


static void FreeNodes(list)
struct List *list ;
{
        while(list->Head->Succ != NULL)
        {
                free(Remove(list->Head)) ;
        }
}
```

```
/* BlobScan.h */


#ifndef STDDEFS_H
#include <stddefs.h>
#endif


#ifndef LIST_H
#include "List.h"
#endif


#define NOEOL            FALSE
#define NUMROWS           480
#define MAXROW            NUMROWS-1
#define NUMCOLS          512
#define MAXCOL           NUMCOLS-1
#define VBLOBLIST_TYPE  1
#define VBLOB_TYPE       2
#define BLOBLIST_TYPE    3
#define ACTUALBLOB_TYPE 4
#define RUNLIST_TYPE     5
#define RUN_TYPE         6


struct Blob
{
        struct Node node ;
        struct VBlob
                *parent,
                *child,
                *sibling ;
        int     number ;
        int     color ;
        int     perimeter ;
        int     NumHoles ;
        int     mincol, maxcol, minrow, maxrow ;
        float   area, sumcol, sumrow ;
        double  sumcol2, sumrow2, sumrowcol ;
        float   xcenter, ycenter ;
} ;


struct VBlob
{
        struct Node node ;
        struct Blob *blob ;
        struct Blob actualblob ;
        int number ;
} ;
```

```
/* BlobScan Enviromnent structure. */
struct BlobScanEnvironment
{
        struct List vbloblist ;
        struct List runlist ;
        struct List bloblist ;
        int NumBlobs ;
        int NumVBlobs ;
        int NumRuns ;
        struct VBlob *CurrentVBlob, *AboveVBlob, *HoleVBlob ;
        struct Run *CurrentRun, *PrevRun ;
        int state, row, col, LeftEnd ;
        BOOL NewBlobFlag ;
        int frame, rowstart, rowend, colstart, colend ;
        int CurrentLine[MAXCOL], PrevLine[MAXCOL] ;
} ;


extern BlobScan(struct BlobScanEnvironment *) ;
extern void CleanUpBlobScan(struct BlobScanEnvironment *) ;
extern struct BlobScanEnvironment
                        *InitBlobScanEnvironment(struct BlobScanEnvironment *) ;
extern struct List *Scan(int, int, int, int, int, int) ;
```

```
/* List.c */


#include <malloc.h>
#include "List.h"


struct Node *InitNode(node, type, pri, name)
struct Node *node ;
UBYTE type ;
BYTE pri ;
char *name ;
{
        node->Type = type ;
        node->Priority = pri ;
        node->Name = name ;
}


struct List *InitList(list, type)
struct List *list ;
UBYTE type ;
{
        list->Head = (struct Node *)&(list->Tail) ;
        list->TailPred = (struct Node *)&(list->Head) ;
        list->Tail = NULL ;
        list->Type = type ;
        return(list) ;
}


struct Node *Insert(t_list, t_node, t_pred)
struct List *t_list ;
struct Node *t_node ;
struct Node *t_pred ;
{
        if(NULL == t_pred)
                t_pred = (struct Node *)&(t_list->Head) ;
        t_node->Pred = t_pred ;
        t_node->Succ = t_pred->Succ ;
        t_pred->Succ->Pred = t_node ;
        t_pred->Succ = t_node ;
}


struct Node *Remove(node)
struct Node *node ;
{
        (node->Pred)->Succ = node->Succ ;
        (node->Succ)->Pred = node->Pred ;
        return(node) ;
}
```

```
struct List *AllocList()
{
        struct List *list ;


        if(NULL != (list = (struct List *)malloc(sizeof(struct List))))
                InitList(list, NULL) ;
        return(list) ;
}
```

```
/* List.h */


#ifndef NULL
#define NULL 0
#endif
typedef unsigned char UBYTE ;
typedef char BYTE ;


struct Node
{
        struct Node *Succ ;
        struct Node *Pred ;
        UBYTE Type ;
        BYTE Priority ;
        char *Name ;
} ;


struct List
{
        struct Node *Head ;
        struct Node *Tail ;
        struct Node *TailPred ;
        UBYTE Type ;
} ;


extern struct Node *InitNode(struct Node *, UBYTE, BYTE, char *) ;
extern struct List *InitList(struct List *, UBYTE) ;
extern struct Node *Insert(struct List *, struct Node *, struct Node *) ;
extern struct List *AllocList() ;
```

VITA

Christine Theresa Rice

Candidate for the Degree of

Master of Science

Thesis: MEASURING CHARACTERISTICS OF RILL EROSION USING IMAGE PROCESSING TECHNIQUES.

Major Field: Agricultural Engineering

Biographical:

Personal Data: Born in Lubbock, Texas, April 23, 1963, the daughter of Harold J. and M. Catherine Altendorf. Married to Christopher K. Rice on December 28, 1985.

Education: Graduated from Putnam City West High School, Oklahoma City, Oklahoma, in May, 1981; received Bachelor of Science Degree in Agricultural Engineering from Oklahoma State University in December, 1985; completed requirements for the Master of Science Degree at Oklahoma State University in December, 1987.

Professional Experience: Research Assistant, Department of Agricultural Engineering, Oklahoma State University, January, 1986 to November, 1987.

Professional Organizations: American Sociey of Agricultural Engineers; National Society of Professional Engineers; Oklahoma Society of Professional Engineers.