AL/HR-TR-1993-0180

VARIABLE-RESOLUTION IMAGERY FOR FLIGHT SIMULATION

George A. Geri

University of Dayton Research Institute
300 College Park Avenue
Dayton, OH 45469-0110

Yehoshua Y. Zeevi

Department of Electrical Engineering
Technion–Israel Institute of Technology
Haifa 32000 Israel

Craig A. Vrana

University of Dayton Research Institute
300 College Park Avenue
Dayton, OH 45469-0110

HUMAN RESOURCES DIRECTORATE
AIRCREW TRAINING RESEARCH DIVISION
6001 South Power Road, Building 558
Mesa, AZ 85206-0904

January 1994

Final Technical Report for Period June 1989 – June 1993

AIR FORCE MATERIEL COMMAND
BROOKS AIR FORCE BASE, TEXAS

ARMSTRONG LABORATORY

NOTICES

This technical report is published as received and has not been edited by the technical editing staff of the Armstrong Laboratory.

When Government drawings, specifications, or other data are used for any purpose other than in connection with a definitely Government-related procurement, the United States Government incurs no responsibility or any obligation whatsoever. The fact that the Government may have formulated or in any way supplied the said drawings, specifications, or other data, is not to be regarded by implication, or otherwise in any manner construed, as licensing the holder, or any other person or corporation; or as conveying any rights or permission to manufacture, use, or sell any patented invention that may in any way be related thereto.

The Office of Public Affairs has reviewed this report, and it is releasable to the National Technical Information Service, where it will be available to the general public, including foreign nationals.

This report has been reviewed and is approved for publication.

Elizabeth L. Martin

ELIZABETH L. MARTIN
Project Scientist

Dee H. Andrews

DEE H. ANDREWS
Technical Director

LYNN A. CARROLL, Colonel, USAF
Chief, Aircrew Training Research Division

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	January 1994	Final – June 1989 – June 1993

4. TITLE AND SUBTITLE
Variable-Resolution Imagery for Flight Simulation

5. FUNDING NUMBERS
C – F33615-90-C-0005
PE – 62205F
PR – 1123
TA – 03
WU – 85

6. AUTHOR(S)
George A. Geri
Yehoshua Y. Zeevi
Craig A. Vrana

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
University of Dayton Research Institute
300 College Park Avenue
Dayton, OH 45469-0110

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAMES(S) AND ADDRESS(ES)
Armstrong Laboratory (AFMC)
Human Resources Directorate
Aircrew Training Research Division
6001 S. Power Road, Bldg 558
Mesa, AZ 85206-0904

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
AL/HR-TR-1993-0180

11. SUPPLEMENTARY NOTES

Armstrong Laboratory Technical Monitor: Dr. Elizabeth Martin, (602) 988-6561.

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

A position-varying, low-pass filter was used to produce variable-resolution images whose spatial frequency content varies as a function of distance from their center. Such images can be matched in some sense to the spatial inhomogeneities of the human visual system, and thus may be visually acceptable even though they contain less information than an unprocessed version of the same image. Following a discussion of image representation with nonuniform sampling and of the concept of locally bandlimited spaces, two experiments were performed to visually assess variable-resolution images. In Experiment 1, images were generated using a series of distortion functions (differing in central blur, peripheral blur, and blur gradient) chosen to approximate cortical magnification functions (CMFNs) derived from existing anatomical and psychophysical data. The distortion functions, corresponding to the images that were just discriminable from an unprocessed image, were used to define the wide-field CMFN associated with changes in blur discrimination across the visual field. In Experiment 2, blur thresholds were measured using apertures in the form of either circles or 3°-wide angular segments centered in either 15° or 30° eccentricity. Differences in blur discrimination for the two stimulus configurations, matched in area, suggest that the spatial organization of the visual mechanisms underlying blur discrimination changes with eccentricity. Finally, the perceptual data obtained here were used to efficiently sample and represent a variable-resolution image.

14. SUBJECT TERMS			15. NUMBER OF PAGES
Cortical magnification factor	Imagery	Visual receptive fields	104
Flight simulation	Nonuniform sampling		16. PRICE CODE
Flight simulation imagery	Variable resolution		

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. Z39-18
298-102

CONTENTS

CONTENTS (cont'd)

List of Figures

PREFACE

The research described here was conducted by the University of Dayton Research Institute (Contract No. F33615-90-C-0005) for the Armstrong Laboratory/Aircrew Training Research Division (AL/HRA) under Work Unit 1123-03-85, Flying Training Research Support. Laboratory contract monitor was Ms. Patricia Ann Spears; task monitor was Dr. Byron J. Pierce. Portions of this research were performed while one of the authors (YYZ) was supported by the USAF-UES Summer Faculty Research Program, Contract No. F49620-88-C-0053.

The authors thank Drs. Elizabeth Martin and Byron Pierce (AL/HRA) for their encouragement and support in all phases of the research reported here, and George Kelly for help in making the MTF measurements. The authors also acknowledge the assistance of Martin Marietta Services, Inc. in maintaining the computers and projector systems used in this research.

VARIABLE-RESOLUTION IMAGERY FOR FLIGHT SIMULATION

SUMMARY

For spatial discrimination tasks, the center of the visual field is more sensitive than the visual periphery. This observation is relevant to the design of visual simulators in that it suggests that information can be removed in the peripheral parts of a simulator image without affecting the quality of the simulation. If this can be done, then smaller, less expensive computer systems can be used to store, manipulate, and transmit imagery at a given level of fidelity. Although there are differences in sensitivity across the visual field, in most cases central and peripheral sensitivity can be equated if the peripheral stimulus is appropriately magnified. Rather than applying spatially varying magnification, we processed images using a spatially varying operator that low-pass filtered (i.e., blurred) the images relatively little at their center and progressively more at greater distances from the center. The result is called a variable-resolution image. A series of such images processed at various levels was used to estimate the blur gradient that was just detectable by human observers. Wide-field (80° diameter), complex, real-world images were used in order to approximate the imagery used in visual simulators. It was possible to obtain (from the variable-resolution imagery) imagery suitable for use in visual simulation since the processing techniques described here are consistent with a formalism which allows appropriately filtered images to be efficiently sampled and hence represented with less information. Based on the variable-resolution functions determined to be near threshold for our observers, we can generate effective variable-resolution visual imagery using, at most, one-quarter of the information associated with conventional imagery.

GENERAL INTRODUCTION

The research described here was designed to perceptually evaluate variable-resolution imagery whose spatial frequency content decreases as a function of distance from its center (Zeevi, Porat, & Geri, 1990). Such imagery is potentially relevant to the design of visual simulators in that it can be made to appear, to a human observer, to be equivalent to an image whose spatial frequency content is equally high at all points. Once an equivalent

variable-resolution image is found, formal procedures (Clark, Palmer, & Lawrence, 1985; Zeevi & Shlomot, 1993) exist for efficiently sampling the image. The more efficient sampled image can then be used in place of the original image in the visual simulator.

We begin here with a brief discussion of conventional, position-invariant image processing techniques in order to distinguish them from the position-varying techniques used in the present study. We discuss some of the characteristics of images processed by our technique, as well as the concept of local bandwidth, and briefly describe a formal technique for appropriately sampling non-uniform imagery. Following this general introduction, two experiments designed to perceptually evaluate variable-resolution imagery are described.

Conventional Image Processing

Conventional techniques of image processing, which are carried out in the context of linear systems theory, are applicable only when the systems used to operate on (or process) the image are *linear and position-invariant*. Under these conditions, it is possible to operate on an image either in the space domain (by convolution) or in the frequency domain (by filtering). For example, operations in the space domain include smoothing an image in order to improve the signal-to-noise ratio, or extracting edges in order to perform pattern recognition or contour identification. The corresponding operations in the frequency domain would be low-pass and band-pass filtering, respectively. The operation, in the space domain, can be represented formally as:

$$l_o(x,y) = \iint l_i(x',y') \cdot k(x-x',y-y')\,dx'\,dy'$$

$$\triangleq l_i(x,y) ** k(x,y) \qquad (1)$$

where $l_i(x',y')$ and $l_o(x,y)$ denote the input and output images, respectively, $k(x-x',y-y')$ is the convolution kernel which represents the mask used to operate on the image, and the double asterisk (**) denotes a two-dimensional convolution. In the spatial frequency domain, the

corresponding expression is obtained by Fourier transforming Equation 1 yielding:

$$L_o(\omega_x, \omega_y) = L_i(\omega_x, \omega_y) \cdot K(\omega_x, \omega_y) \quad (2)$$

where L_o and L_i are the spatial frequency spectra of the input and output images, respectively, and $K(\omega_x, \omega_y)$ is the modulation transfer function (MTF) of the system. An example of an image generated using a discrete implementation of the convolution operation of Equation 1 is shown on the right in Figure 1. The original unprocessed image is shown on the left. The convolution was performed with a 39 x 39-pixel gaussian kernel. The summation kernel was position-invariant and thus the image has been low-pass filtered (i.e., blurred) equally at all positions.

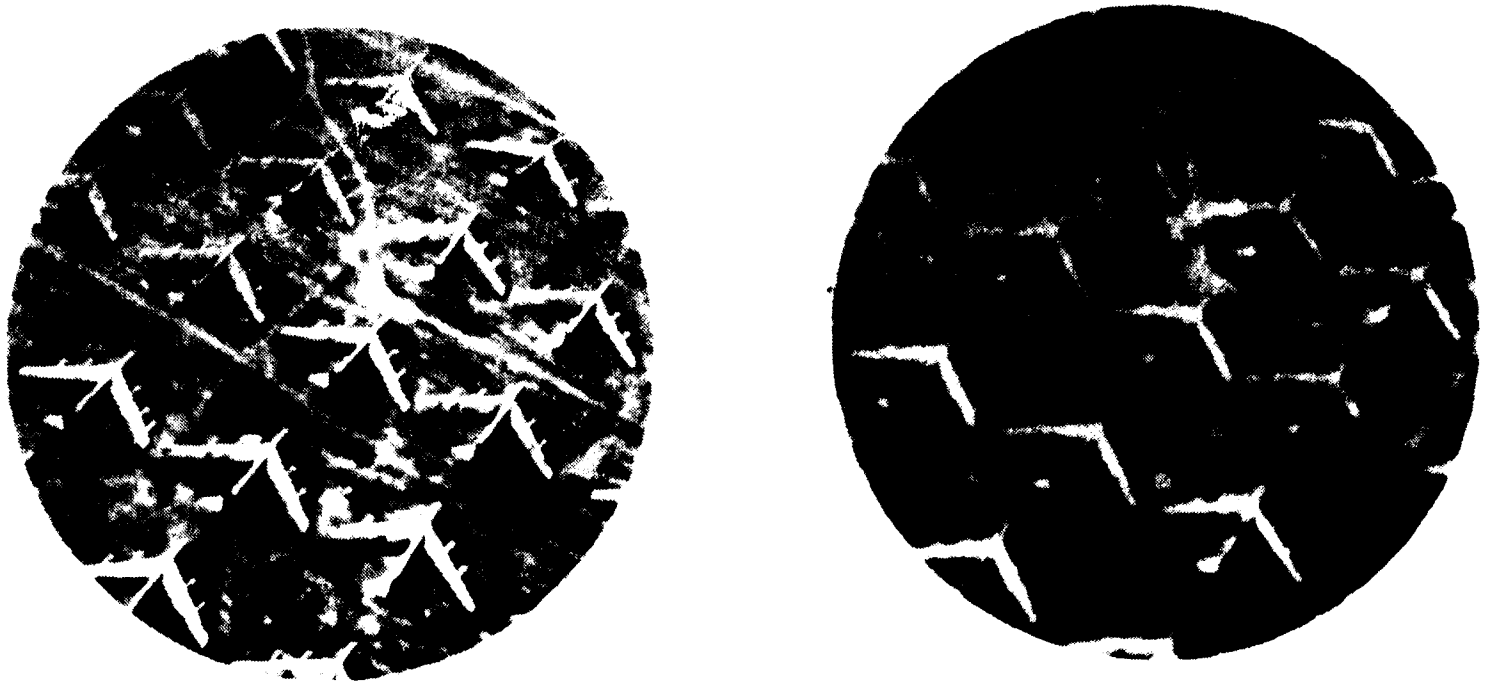
Variable-Resolution Imagery

In order to process images in a *position-varying* fashion similar to that taking place in the visual system, we cannot, for the reasons described above, apply the classical techniques of linear systems theory. It is necessary, therefore, to devise techniques which are analogous to convolution and other linear operations, but which are not position-invariant. The technique we have chosen involves operating on a fixed-resolution image with a system whose characteristics vary from point to point. Thus, the kernel, $k(x-x',y-y')$, shown in Equation 1, which is a function only of the differences between pairs of variables along the two spatial coordinates, becomes $k(x,x',y,y')$, which is a function of the two stimulus independent variables and the two response independent variables. Thus, in this case:

$$l_o(x,y) = \iint l_i(x',y') \cdot k(x,x',y,y')\,dx'\,dy' \quad (3)$$

In accordance with Zeevi, Peterfreund, and Shlomot (1988), we will refer to the function $l_o(x,y)$ as a *variable-resolution image*. An example of a variable-resolution image is shown on the right in Figure 2. Again, the original unprocessed image is shown on the left. The image was processed using a discrete implementation (see Methods, Experiment 1) of the integral operation shown in

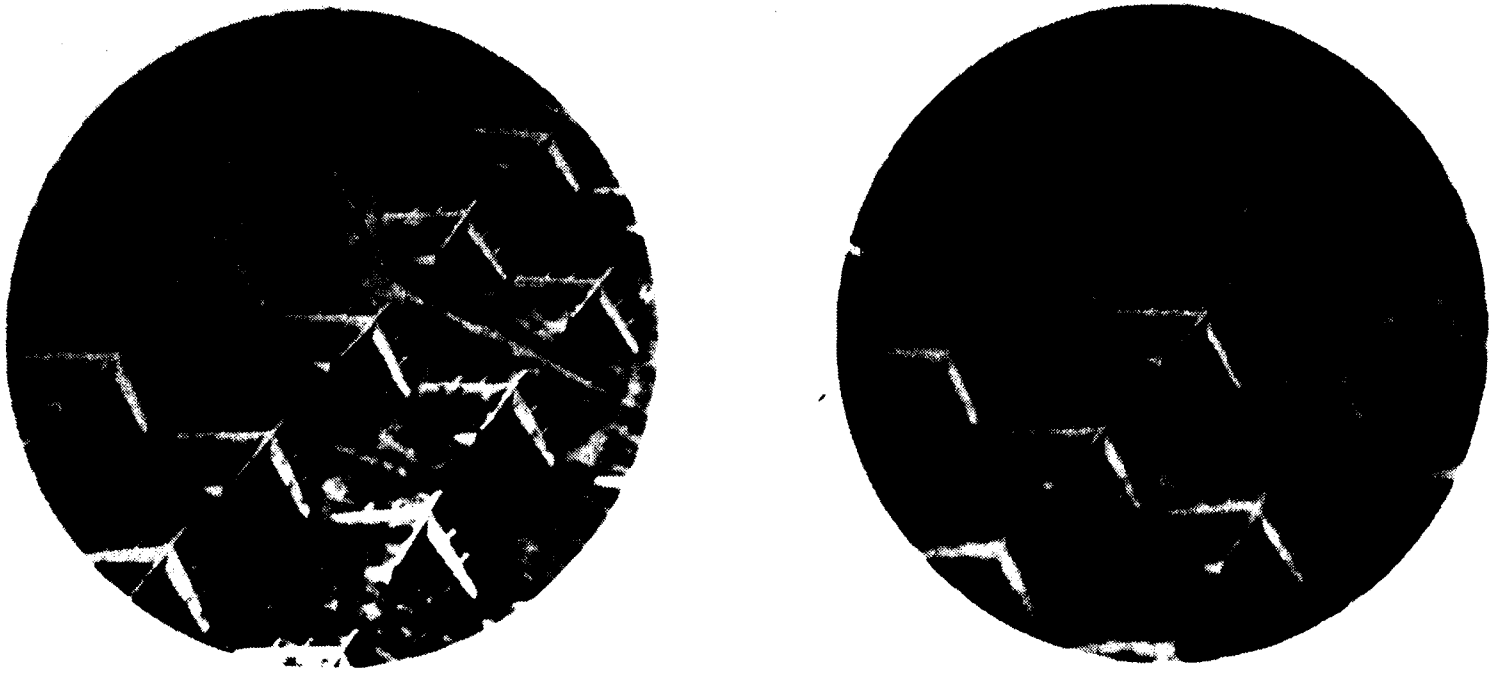
Convolution



$$I_o(x, y) = \iint I(x', y') \cdot k(x-x', y-y')\, dx'dy'$$

Figure 1
Low-Pass Filtering Using a Position-Invariant Gaussian Kernel

Variable Resolution



$$I_o(x, y) = \int\int I_i(x', y') \cdot k(x,x', y,y')\, dx'dy'$$

Figure 2
Variable Resolution Produced Using a Position-Varying Gaussian Kernel

Equation 3. The processing was performed with a gaussian kernel whose size varied from 1 x 1 pixels, at the center of the image, to 49 x 49 pixels at the edge of the image.

Since Equation 3 is not of the form which defines a convolution, the Fourier transform cannot be applied to it, and thus an MTF cannot be defined for the system (filter) which was used to operate on the image. Since it is not possible to condition the spectral distribution of an image by multiplying it by the transfer function of the system, it is always necessary to operate in the space domain by solving the integral equation which represents the operation of a filter whose form depends on position.

Unlike the convolution operation of Equation 1, which can be performed only in the context of linear, position-invariant systems, the integral operation described by Equation 3 may transform a bandlimited signal into a signal whose bandwidth is not limited. In the case of systems which are linear and position-invariant, this cannot occur as can be concluded most easily from an examination of the operation in the frequency domain. Since the convolution integral is transformed into the product of the spectrum of the input signal and that of the MTF of the system, it is clear that no extra frequency components can be introduced. It is well known, however, that nonlinear systems introduce frequency components which are not present in the input signal. Extra frequency components or bandwidth expansion can also result from a superposition integral of the type shown in Equation 3, which represents a linear but position-varying operation.

Bandwidth expansion of an input signal by the addition of frequency components is more often encountered in the case of nonlinear operations (i.e., nonlinear systems). However, this type of bandwidth expansion can also result from the operation of a system which is linear but *position-varying*. Consider a signal which is bandlimited and thus satisfies the Nyquist condition. If the signal is transformed by distorting the position axis, the resulting signal no longer satisfies the Nyquist condition in that the signal is no longer bandlimited. The process of distortion is analogous in a way to FM modulation of a bandlimited signal which introduces extraneous frequency components (in fact, extends the band to infinity). The distorted signal, therefore, cannot be represented by a discrete set of samples.

Locally-Bandlimited Imagery

According to classical sampling theory (Jerri, 1977), a signal (or image) must be convolved with a sinc function (low-pass filtered) prior to being sampled so that its bandwidth is appropriately (inversely) related to the required sampling interval. This filtering operation projects the image into a space of *bandlimited functions* (BLFs) so that it can be represented using a finite sampling rate. There are infinitely many such spaces, denoted B^{Ω}, each defined by its associated bandwidth, Ω (see e.g., Zeevi & Shlomot, 1993).

There are certain conditions, however, under which an image can be represented by a finite number of samples even if it is not bandlimited. For example, if we had begun with a bandlimited image and applied our variable-resolution procedure to it, the resulting image would no longer be bandlimited. However, because the underlying distortion function, used to alter the spatial distribution of information in the image, is of a form for which an inverse exists (cf., Clark, *et al.*, 1985), that inverse can be used to restore the image to one whose information is distributed uniformly and which is again bandlimited. In this case, the generalized sampling procedure can be applied and the resulting variable-resolution image can be completely represented by a finite set of samples. Images which are not bandlimited but can be converted to bandlimited images by the application of the inverse of a distortion function are said to belong to the space of *locally-bandlimited functions* (LBLFs). This space is denoted by B_{γ}^{Ω}, where Ω is now the local bandwidth (see below), and γ corresponds to the associated distortion function.

As noted earlier, in the case of variable-resolution images the integration kernel is *spatially varying* and therefore the associated integral operation (Equation 3) is not a convolution. Nevertheless, assuming that the integration kernel is a sinc-function, this operation projects images onto a well-defined space of LBLFs (Zeevi & Shlomot, 1993). Analogous with the bandwidth which defines each space of BLFs, there is a measure, called *local-bandwidth*, which defines each space of LBLFs. The concept of local bandwidth may at first appear to be ill-defined in that it seems to be self-contradictory according to the uncertainty principle. According to this principle, in order to have a limited bandwidth, the function has to be of infinite extent and the information regarding the bandwidth (which determines in turn the resolution in frequency) is determined in relation to the entire extent of the signal. The term local bandwidth implies that the frequency characteristics are derived from local properties, hence the

contradiction. It is therefore important to elaborate on this issue. The concept of local bandwidth can best be understood in the context of signal representation in the combined position-frequency space where a different bandwidth can be specified for each position (within the limitations imposed by the joint uncertainty; Gabor, 1946). Thus, one interpretation of local bandwidth is that the local properties of the image can be used to determine a transformation of it which will produce a globally bandlimited image (i.e., one that belongs to B^{Ω}). Another interpretation of the local bandwidth can be understood in the context of a position-varying MTF analogous to the one introduced by Zadeh (1952) in his analysis of time-varying systems. In analogy to the formalism proposed by Zadeh, the MTF is a function of both position and frequency, and as such, one can formally define the local bandwidth for a given position. Similarly, Horiuchi (1968) proposed that if a signal can be obtained by the inverse Fourier-like operation defined as follows:

$$f(x) = \frac{1}{2\pi} \int_{-\pi W(x)}^{\pi W(x)} F(x, \omega) \cdot \exp(i\omega x)\, d\omega \tag{4}$$

where $F(x,\omega)$ is a position-dependent, Fourier-like, transform of $f(x)$, then the signal has a position-varying bandwidth $W(x)$. Zeevi and Shlomot (1993) extended the concept of position-varying bandwidth by showing that any signal which belongs to B_{γ}^{Ω}, is also characterized by a local bandwidth.

Once an analogy is drawn between uniform and nonuniform operations, the concepts of "local bandwidth" and "B_{γ}^{Ω}" (and their application to the generation of variable resolution imagery) become relatively simple and straightforward. It should be stressed that the proposed technique for projecting a given image into B_{γ}^{Ω} is the proper way of dealing with the issue of variable resolution in that the resultant local properties are well-defined, and in that such an image can be represented by a (finite) set of samples and reconstructed without any aliasing effects even though it is not bandlimited and as such does not satisfy the Nyquist condition. Further, such a technique permits additional nonuniform processing in the form of nonuniform pyramids and other schemes (Peterfreund & Zeevi, 1992; Zeevi *et al.*, 1988).

Sampling Nonuniform Images

If an image has been transformed (e.g., into a variable-resolution image) by a known

distortion function, there are in principle two ways by which it can still be represented by adiscrete set of samples. First, the signal can be restored (undistorted) and then represented like any other bandlimited signal. Image reconstruction is then performed in accordance with the provisions of the Whittaker-Shannon sampling theorem using an interpolation filter in the form of a sinc function (cf., Jerri, 1977). Stated another way, if an image can be projected into the space of BLFs by a transformation along the spatial axis, then it can be adequately represented by a discrete set of samples. The second way of representing a transformed image by a discrete set of samples is to distort and position the interpolation functions (sinc functions) nonuniformly in accordance with the positional distortion function, and then represent the signal even though it is not bandlimited and as such does not satisfy the Nyquist condition (Clark *et al.*, 1985; Zeevi & Shlomot, 1993).

If the distortion function, as described above, is not known, or if it is not known whether the image even belongs to the space of LBLFs, a number of theoretical and practical questions arise. For example, if it is not known what type of nonuniform processing was used on a given bandlimited image to generate a variable resolution image, how can it be determined whether the latter belongs to any space of LBLFs and, if so, to which of the infinite number of such spaces? Theoretically, in order to determine whether an image belongs to a particular space of LBLFs, one would have to first apply the nonuniform filter which would project that (or any other) image into that space. If the resultant image is identical to the original one, this implies that the image belonged to the space to start with, and that the space itself is a so-called *reproducing kernel space* (Aronszajn, 1950). Since there are infinitely many such spaces, it is not possible of course to proceed in this way. The alternative, practical, approach is to devise techniques for estimating a distortion function which optimally matches the nonuniform distribution of information over the image field. The techniques might be based on estimating statistically the rate of zero-crossings from a number of images, or by estimating the effective local bandwidth which characterizes the representation of the image in some combined position-frequency space, such as those based on the Gabor scheme, the Wigner distribution, or the complex spectrogram (cf., Zeevi & Shlomot, 1993). Estimation of the optimal distortion function will determine which of the infinite number of locally bandlimited image spaces the given image fits best in the sense of having a minimal distance from the original image according to some criterion. Several techniques have been

proposed by Zeevi and Shlomot (1993) for estimating the distortion function which best represents a given image in that once the corresponding projection to the (well-defined) space of LBLFs is performed, a minimal least-mean-square error results.

In the context of the present study, the problems are somewhat simpler because either the exact distortion function, corresponding to a given image, is known or the type of nonuniform processing applied to a bandlimited image is known and, as such, can be used in estimating the distortion function. As a practical matter, we need to determine how to nonuniformly distribute the sampling points such that the interpolated variable resolution image will be transformed into an LBLF. Starting with a standard (uniform-resolution) image, how can one devise a nonuniform filter which will result in the desired distribution of resolution across the image while at the same time providing a formalism which satisfies certain theoretical constraints related to the proper representation of the image by a set of sampling points and the reconstruction of the image from that set with no aliasing effects?

Sampling Locally Bandlimited Images

The purpose of development of the variable resolution technique is its eventual application in efficient image representation in flight simulators (and other types of high-fidelity, wide field-of-view display systems). Thus, for the purpose of image storage and/or manipulation, it is desired to exploit the variable-resolution properties of the image such that the data set can be reduced accordingly. In fact, this is the very reason for generating variable resolution imagery. In some flight simulators, computer-generated images are stored and manipulated as a discrete set of points, (i.e., a two-dimensional array of samples). The question then is, how should the sampling points be nonuniformly distributed over the image field such that they adequately represent the image in the sense that the variable resolution image can be reconstructed from this set of sampling points. Intuitively, it is clear that as the data become more sparsely distributed as a result of variable resolution processing, the required density of sampling points decreases. A sampling theorem for images which belong to B_γ^Ω was first introduced by Clark, *et al.* (1985). This theorem was extended by Zeevi and Shlomot (1993) to allow images to be sampled using other than rectilinear patterns. As stated earlier, Zeevi and Shlomot also showed that the kernel which projects images into B_γ^Ω is a reproducing kernel. A formalism for reconstructing an image,

$f(\mathbf{x})$, from a properly chosen set $\{\mathbf{x}_n\}$ of nonuniformly distributed samples will now be described.

To highlight the analogy that exists between uniform and nonuniform sampling schemes and to better understand the latter, we consider first the case of uniform sampling. Let $g(\mathbf{x})$ belong to the space B^{Ω} (i.e., assume that the area of support, Ω, defined by its spectrum is limited), and let the periodicity matrix, $\mathbf{U}$, represent the periodic extension of the spectrum, $G(\omega)$, associated with the sampling operation. Then, $g(\mathbf{x})$ can be reconstructed (interpolated) from its samples along the uniform grid, $\mathbf{x}_n = \mathbf{V}\cdot\mathbf{n}$, [where the sampling matrix, $\mathbf{V}$, satisfies the condition that the inner product of $\mathbf{U}$ and $\mathbf{V}$ equals a constant times the identity matrix, (i.e., $\mathbf{U}^t\mathbf{V} = 2\pi\mathbf{I}$)], according to the following formula:

$$g(\mathbf{x}) = |\det \mathbf{V}| \cdot \sum_n g(\mathbf{x}_n) \cdot \phi(\mathbf{x} - \mathbf{V}\mathbf{n}) \tag{5}$$

where $g(\mathbf{x}_n)$ represent amplitudes of the interpolation function, $\phi(\mathbf{x})$, defined by,

$$\phi(\mathbf{x}) \stackrel{\Delta}{=} \frac{1}{4\pi^2} \iint_{\Omega} \exp(i\omega^t\mathbf{x})\, d\omega \tag{6}$$

Thus $\phi(\mathbf{x})$ represents the Fourier transform of a rectangular pulse of unit height and dimension Ω x Ω (i.e., a 2-D sinc function).

We now extend the above theorem to include functions which belong to B_{γ}^{Ω}. If $g(\mathbf{x})$ is a *globally* BLF (i.e., if $g(\mathbf{x}) \in B^{\Omega}$), then $f(\mathbf{x})$ is a LBLF (i.e., $f(\mathbf{x}) \in B_{\gamma}^{\Omega}$), if and only if, a distortion function, $\gamma(\mathbf{x})$, exists such that $f(\mathbf{x}) = g[\gamma(\mathbf{x})]$ (i.e., such that $f(\mathbf{x})$ can be expressed as a function of g, and hence is also a member of B^{Ω}). Note that if $\gamma(\mathbf{x}) = \mathbf{x}$, then the two above-described spaces become identical. Such an LBLF can be reconstructed from the properly distributed set of samples, $\{\mathbf{x}_n\}$, where $\mathbf{x}_n = \gamma^{-1}(\mathbf{V}\cdot\mathbf{n})$, according to the following formula:

$$f(\mathbf{x}) = g[\gamma(\mathbf{x})] = |\det \mathbf{V}| \cdot \sum_n g(\mathbf{V}\mathbf{n}) \cdot \phi[\gamma(\mathbf{x}) - \mathbf{V}\mathbf{n}$$

$$= |\det \mathbf{V}| \cdot \sum_n f(\mathbf{x}_n) \cdot \phi[\gamma(\mathbf{x}) - \mathbf{V}\mathbf{n}) \tag{7}$$

which represents interpolation using "uniformized" samples. Note that the effect of $\gamma(x)$ was to change the distance between pulses resulting in nonuniform sampling and a partial overlap of $G(\omega)$. The resulting function is not bandlimited and hence aliasing could occur.

The implementation of the above formalism can be better understood by considering the reconstruction of a one-dimensional signal in which case Equation 7 becomes:

$$f(x) = \sum_{n=-\infty}^{\infty} f(x_n) \cdot \mathrm{sinc}[\gamma(x) - n] \tag{8}$$

where $\mathrm{sinc}(x) \triangleq \sin(\pi x)/\pi x$.

Substituting $\gamma(x) = W(x) \cdot x$ into Equation 8 yields:

$$f(x) = \sum_{n=-\infty}^{\infty} f(x_n) \cdot \mathrm{sinc}[W(x) \cdot x - n] \tag{9}$$

where $W(x)$ provides a measure of the sampling density and of the local bandwidth. As noted above, if $\gamma(x) = x$, the function $f(x)$ belongs to B^{Ω}. Indeed, in that case, $W(x) = 1$ and we have uniform sampling density, and Equation 9 becomes the standard equation of interpolation from a uniformly distributed set of samples.

EXPERIMENT 1. PERCEPTUAL ASSESSMENT OF WIDE-FIELD, VARIABLE-RESOLUTION IMAGERY

INTRODUCTION

The well-documented variations in spatial discrimination across the visual field (Levi, Klein, & Aitsebaomo, 1985; Rovamo & Virsu, 1979; Virsu, Näsänen, & Osmoviita, 1987) suggest that conventional, position-invariant processing techniques cannot produce images which are efficiently matched to the human visual system. In order to match the spatial inhomogeneities of the human visual system, images must be processed in a position-varying manner. This would

be necessary, for instance, to effectively implement high resolution over a wide field of view given limited computational and bandwidth resources (Zeevi *et al.*, 1990). Techniques have recently been developed (Peterfreund & Zeevi, 1990; 1992; Zeevi & Shlomot, 1993; Zeevi *et al.*, 1988) for filtering images by position-varying techniques in a manner analogous to the low-pass filtering performed prior to sampling with fixed sampling rate (i.e., position-invariant sampling).

As noted above, the information processing capability of the human visual system decreases from the center of the visual field to the visual periphery. A so-called cortical magnification factor (CMF) can be specified which relates the sensitivity at a given retinal eccentricity to that at the fovea. The CMF is presumed to reflect the relatively greater number of cortical cells associated with a given retinal area near the fovea as compared to an equivalent area in the periphery (Daniel & Whitteridge, 1961; Talbot & Marshall, 1941; Van Essen, Newsome & Maunsell, 1984). Thus, it follows that, for those visual tasks that can be scaled, the number of cortical cells stimulated can be equated for stimuli in the center and periphery if the latter are appropriately magnified.

In Experiment 1, test images were generated using a position-varying filter whose bandwidth decreased as a function of distance from the center of the image. The specific functions tested were chosen to approximate the observed variation in the CMF across the visual field (Rovamo & Virsu, 1979). There are disagreements in the literature as to the specific function that relates CMF to eccentricity (Dow, Snyder, Vautin & Bauer, 1981; Levi *et al.*, 1985; Rovamo & Raninen, 1984), but this issue is not addressed here. Rather, several candidate linear functions were chosen that differed in central blur (at 8° eccentricity), peripheral blur (at 40° eccentricity), and the blur gradient between 8° and 40° eccentricity. The function that produced a blur gradient which was just discriminable from an unprocessed image was taken as the distortion function which may in turn be used to efficiently sample (i.e., subsample) real-world images such as those used in visual simulators. Although the processing performed here is not strictly an image magnification, there is some analogy in the change in spatial frequency content of the images resulting from these two procedures. Thus, given that formal procedures have been developed for sampling and generating variable-resolution imagery (Clark *et al.*, 1985; Zeevi *et al*, 1988), this imagery may be most useful for producing efficient visual simulation.

METHOD

Generation of Variable-Resolution Images

A locally bandlimited variable resolution image can be produced only by using a projection technique, similar to that described by Equation 3, wherein a (variable) sinc function is used as the integration kernel. Such a technique is computationally intensive and relatively difficult to implement, we have, therefore, chosen an alternative technique which uses an integration kernel based on a gaussian function whose space constant varied with position in the image. Although this approach does not conform to any complete formalism, which might otherwise be used to fully characterize the resulting image for the purposes of further processing and representation, it is sufficient for obtaining preliminary psychophysical data.

Because we desire the final variable-resolution image to be radially symmetric (i.e., to display the same amount of degradation at all points equidistant from the center), it is convenient to first transform the original image from cartesian to polar coordinates before applying the various distortion functions that determine how the low-pass characteristics change across the image space. In fact, the polar-coordinate representation is inherently appropriate for generating variable-resolution imagery in that it allocates a larger area to central areas of the cartesian version of the image than to peripheral ones. Also, it presents the image data points in an ordered manner that makes the integration (or summation in the discrete case) more convenient. After the integration operation was performed, the image was transformed back into the cartesian coordinate system. Variable-resolution images were then generated using a discrete implementation of the integral operation described by Equation 3:

$$l_o(x,y) = \sum_m \sum_n l_i(x',y') \cdot k(x,y,x',y') \tag{10}$$

where k is the gaussian summation kernel (see Appendix A). The width of the kernel varied along both the radial and angular coordinates and was determined by sampling a function of the form:

$$k(x,x',y,y') = \exp\left[-\left[\frac{(x-x') + (y-y')}{D}\right]\right] \quad (11)$$

where x' and y' are the coordinates of the image pixel over which the summation kernel was centered, and D is the effective width ($\pm 1\sigma$) of the gaussian. The summation kernels varied in size from 3 x 3 to 13 x 13 pixels. Following the summation operation, the image was transformed back into the cartesian coordinate system. A transformation from cartesian to polar coordinates and back may result in approximation errors since sampling is discrete and hence not all transformed points can be accurately represented in both coordinate planes. At the level of resolution used in the present study, the transformation did not generate any noticeable image degradation which might have been mistaken for variable resolution.

The linear functions shown in Figure 3 were chosen to vary the parameters which specified the integration kernel as a function of distance from the center of the image. These functions were chosen to approximate, to varying degrees, the cortical magnification functions (CMFNs) derived from anatomical and psychophysical data (Dow *et al.*, 1981; Rovamo & Virsu, 1979). The full set of functions shown in Figure 3 allows an assessment of central blur, peripheral blur, and blur gradient, although all of these factors were not tested independently of the others. Linear functions were considered acceptable since they give a good approximation to published CMFNs. For instance, between 8° and 40° eccentricity, the CMFN suggested by Rovamo and Virsu is nearly linear and varies from 7 to 9 (corresponding here to our kernel size) over that range.

The stimuli used in the present study extended from 8° to 40° eccentricity. The central portion of the stimulus was removed for two reasons. First, the CMFN changes rapidly near the fovea and the available display resolution was not high enough to give an accurate representation in that portion of the visual field. Second, it was expected that observers would find it difficult to distribute their attention over a stimulus area that was 80° in diameter. This would have been especially difficult if they were asked also to attend to visual detail presented at or near the fixation point.

Observers

The observers were two males (SF and GG) and one female (KV) who were 25, 41, and

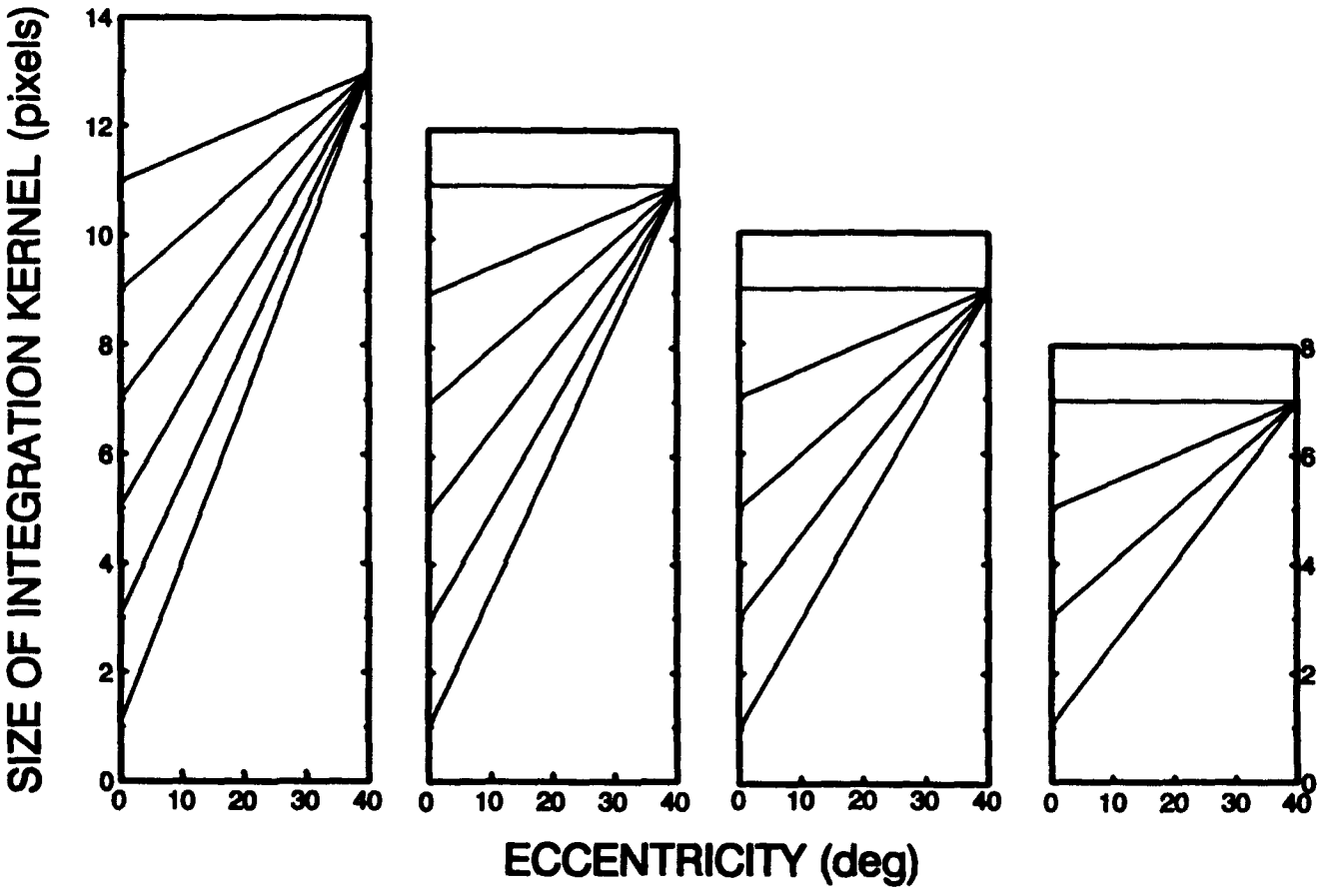


Figure 3
Variable-Resolution Functions Tested in Experiment 1

24 years of age, respectively. All observers had normal uncorrected vision. Observers SF and KV were paid for their participation, while observer GG was one of the authors.

Stimuli and Apparatus

Two real-world aerial photographs were digitized, and four stimulus templates were constructed by pairing each half of the two photographs with its mirror image and removing a portion of the center of the image. One-half of each stimulus was then low-pass filtered (i.e., blurred) in accordance with one of the functions of Figure 3. Two of the stimulus templates, corresponding to one blur gradient (much higher than those used in the present study) applied to the left half of each image, are shown in Figure 4. The programs used to produce the stimulus templates for both Experiments 1 and 2 are presented in Appendix B. The full stimulus set consisted of 108 images. Of these, 84 were test stimuli corresponding to the four image templates each processed at one of the 21 filter gradients. The remaining 24 stimuli were control stimuli consisting of each image template processed on both sides by one of seven (i.e., 1 x 1 through 13 x 13) kernels. An equal number of test and control stimuli were presented in each session. The functions in only one panel of Figure 3 were tested in a given session, which included presentation of all four image templates. In a given session, the test stimuli were presented with the unprocessed image appearing on the right 10 times and on the left 10 times. Each control stimulus was accordingly presented 20 times per session since no distinction could be made between the left and right halves of these stimuli. Between two and four sessions were run for each observer under the four conditions corresponding to panels of Figure 3. All stimuli were presented 1m from the observer and they extended radially for 40° from a fixation point located at the center of the image. Stimulus duration was 167 msec. The observer was seated and used a chin and head rest to maintain a constant position relative to the screen.

All image presentation and data collection were under the control of a Silicon Graphics Iris workstation. The stimuli were presented using the green channel of a Sony Superdata CRT projector (Model VPH-1270Q), and a rear-projection screen (Lumiglass 350, Stewart Film Screen Corp.). Subject responses were made using a mouse interfaced with the computer. Display luminance was measured using a Photo-Research Model 1500 photometer. The gamma-function for the green CRT channel was measured and linearized using an 8-bit look-up

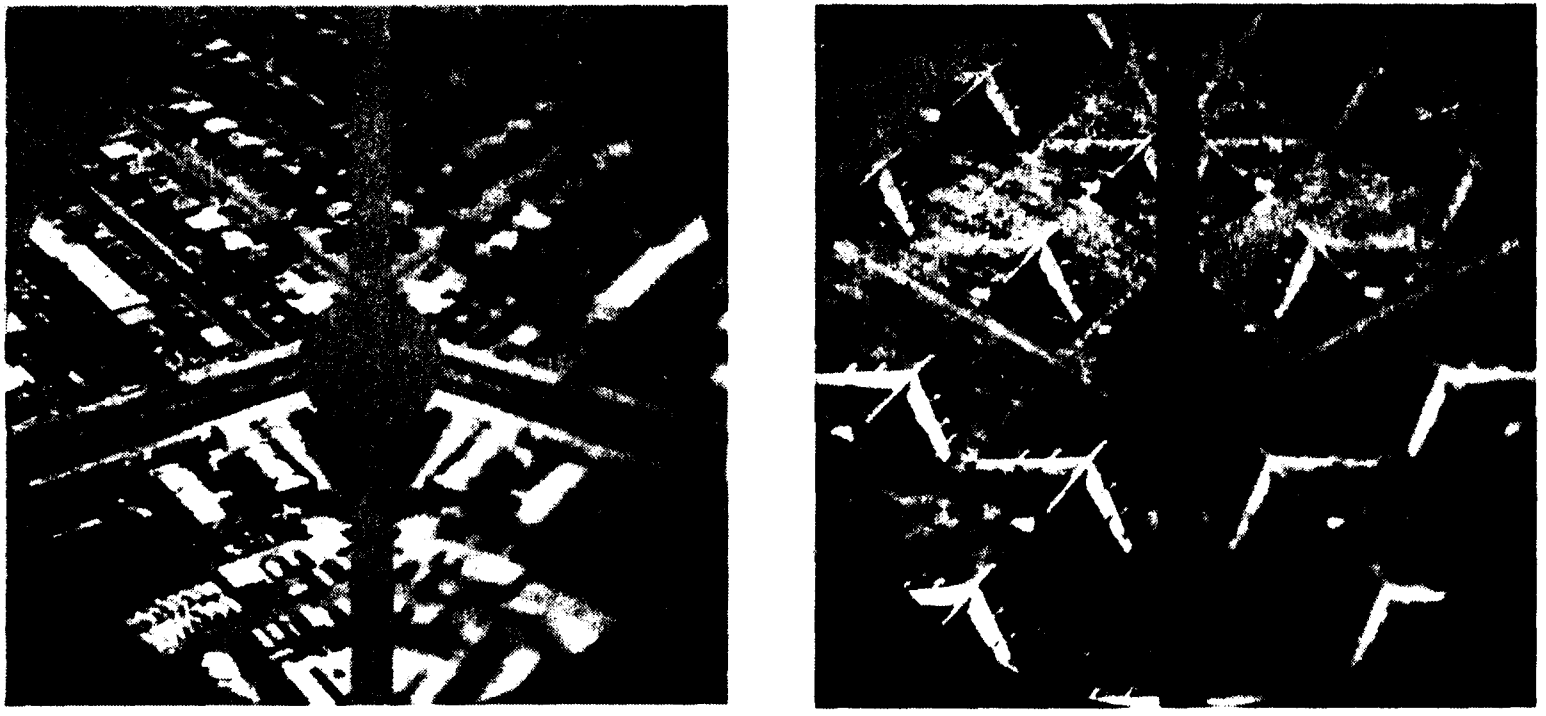


Figure 4
Two of the Image Templates Used in Experiment 1

table. In addition, a luminance measurement was obtained at 24 equally spaced points between the center and edge of the display area in order to correct for the luminance gradient caused by the projector optics and the directional properties of the rear-projection screen. The luminance gradient was corrected off-line by appropriately adjusting the 8-bit gray scale of each stimulus image. The mean luminance of the stimuli was 0.5 fL.

A set of MTFs for the display system was calculated from horizontal and vertical line-spread functions associated with lines of lighted pixels located at the center of the display and at points 12° and 28° from the center in both directions along the horizontal meridian. Measurements were made using a Photo-Research (Model PR-719) Spatial Scanner, and software (see Appendix C) that implemented the procedures for calculating MTFs described by Kelly (1992). The horizontal and vertical MTFs calculated for the display system are shown in Figures 5a and 5b, respectively. The horizontal MTFs (each calculated from a vertical line of pixels) were virtually identical at all points tested across the screen. There were, however, differences in the vertical MTFs for those same points. As a result, a residual blur gradient was effectively added to all horizontally oriented stimulus features. However, as the data of Figure 5b show, this gradient was relatively small and in a direction opposite to that of the gradients evaluated in the present study. Further, in the present study, an unprocessed image was presented on all trials, and because both the horizontal and vertical MTFs measured at a given distance from the center of the image were similar, no differential response would be expected based on the measured differences in display MTF. The data presented here should, however, be taken as conservative estimates of the observers' sensitivity to image blur since higher sensitivity would be expected had the residual blur been removed.

Procedure

Each experimental session began with a 5-10 min period of adaptation to the low ambient illumination of the experimental room. The observers then viewed, for 2-3 min, the center of an illuminated screen, which was the same size and had the same mean luminance as the experimental stimuli. The observer initiated the experimental session using a mouse, and the first stimulus appeared within 3-4 sec. The observers were asked to respond as to which half of the image appeared to be more blurred. There were 3-4 sec between the observer's response and the next

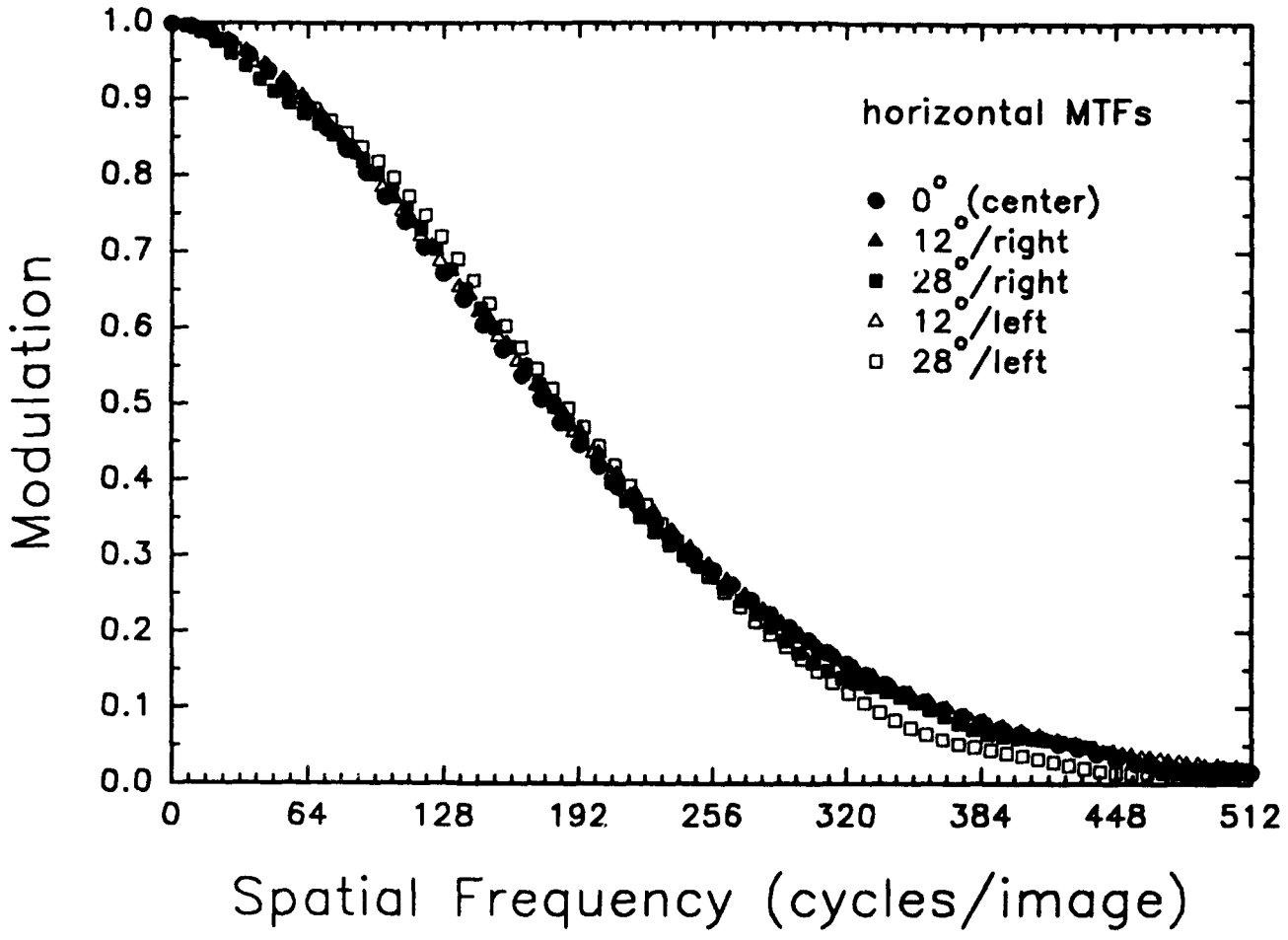


Figure 5a
Horizontal MTFs of the Stimulus Display System

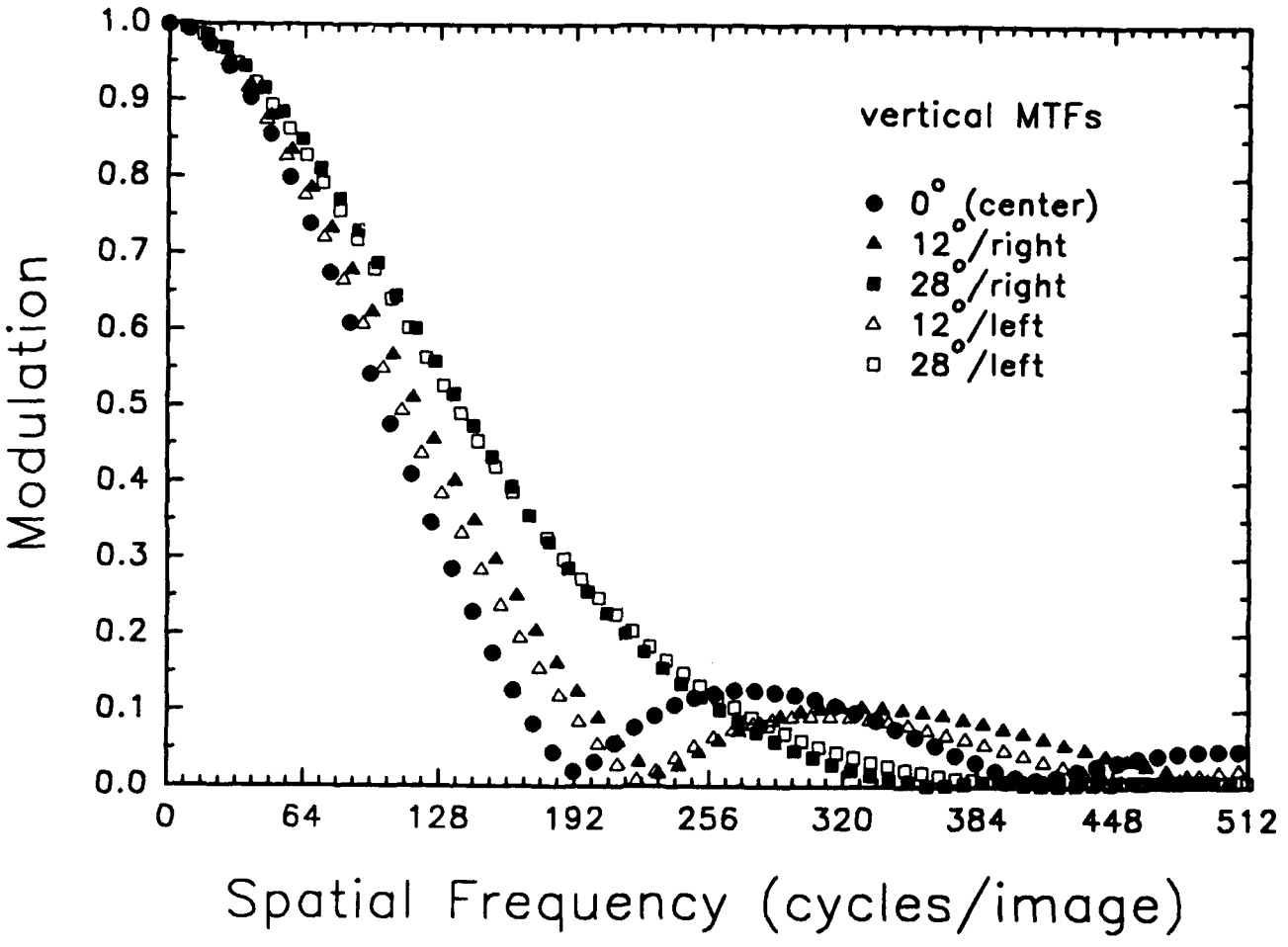


Figure 5b
Vertical MTFs of the Stimulus Display System

stimulus. The experimental session consisted of between 320 and 480 stimulus presentations broken up into three subsessions each consisting of as close to one-third of the trials as possible. The observers controlled the amount of time between each subsession and could also initiate rest periods within the subsessions by delaying their response. A mean-luminance screen was present between stimuli and during all rest periods.

RESULTS

The discrimination data obtained from the three observers are summarized in Figures 6a, 6b, and 6c, respectively. The data for both the bomber and city images were similar and so all data were pooled for each observer. The four plots associated with each observer correspond to the four sets of variable-resolution functions tested. Based on the percentage of correct responses (%C), three levels were used to categorize the observers' ability to discriminate an unprocessed image from an image processed according to each of the functions within each set. The dark-shaded areas of Figure 6 encompass the variable-resolution functions that were discriminated more than 85% of the time, while the white areas encompass the functions that were discriminated at near the chance level (<65%C). The functions discriminated between 65% and 85% of the time were defined as being at or near threshold, and are identified by the gray-shaded areas in the figure.

The data of Figure 6 show that observer SF was able to discriminate the low-pass filtered images somewhat better than observer GG who, in turn, was somewhat more sensitive than observer KV. Despite these quantitative differences, all observers showed qualitatively similar patterns of sensitivity across the four sets of stimuli tested. For instance, whereas discrimination performance declined as the size of the integration kernel applied to the peripheral edge of the image was decreased, the decline could be compensated by a (generally smaller) increase in the size of the kernel used at the more central edge of the image.

DISCUSSION

The second function from the top in the third panel of Figure 3 most closely approximates the human CMFN suggested by Rovamo and Virsu (1979). When the data for all three observers

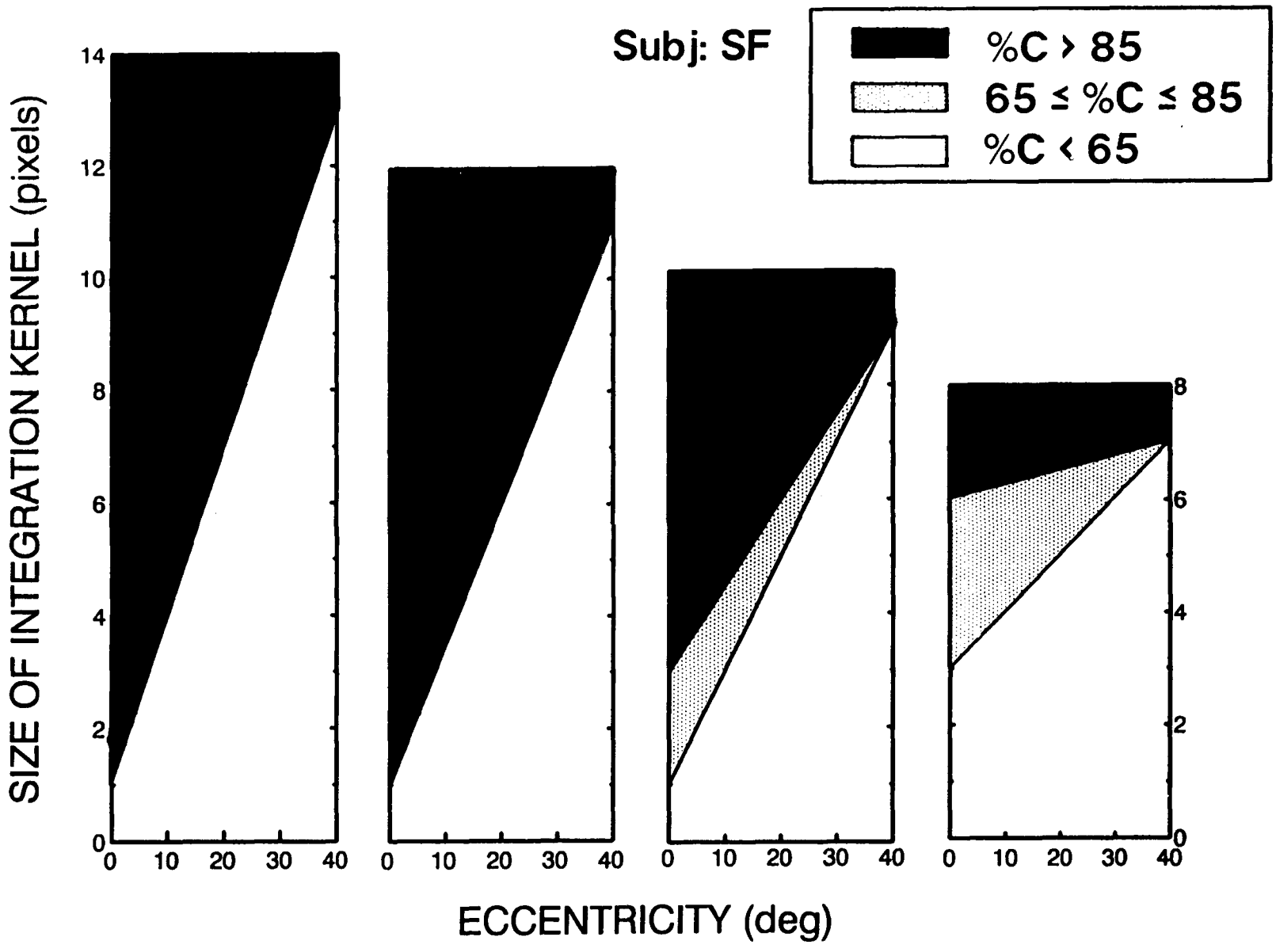


Figure 6a
Blur Discrimination Data for Observer SF

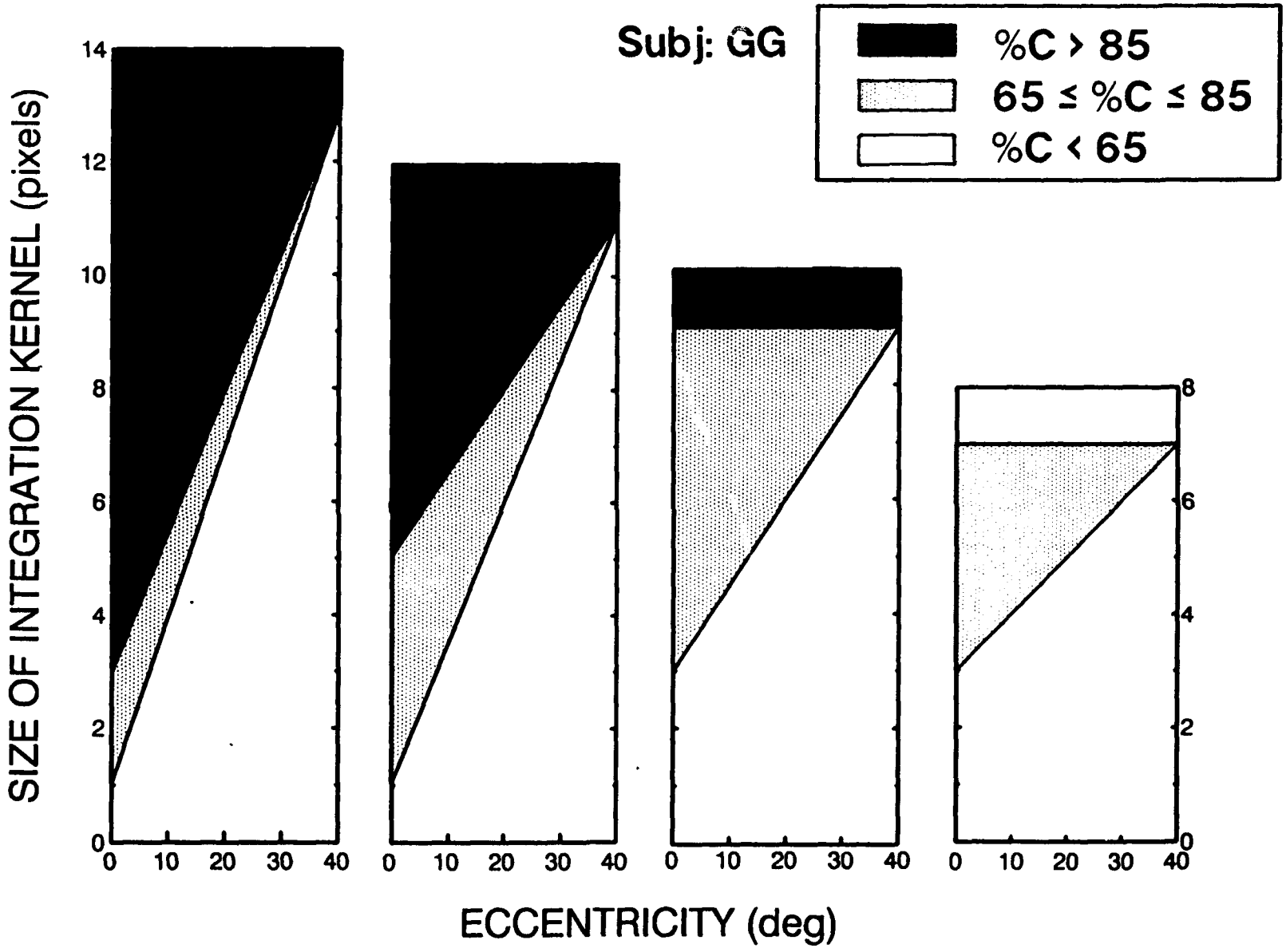


Figure 6b
Blur Discrimination Data for Observer GG

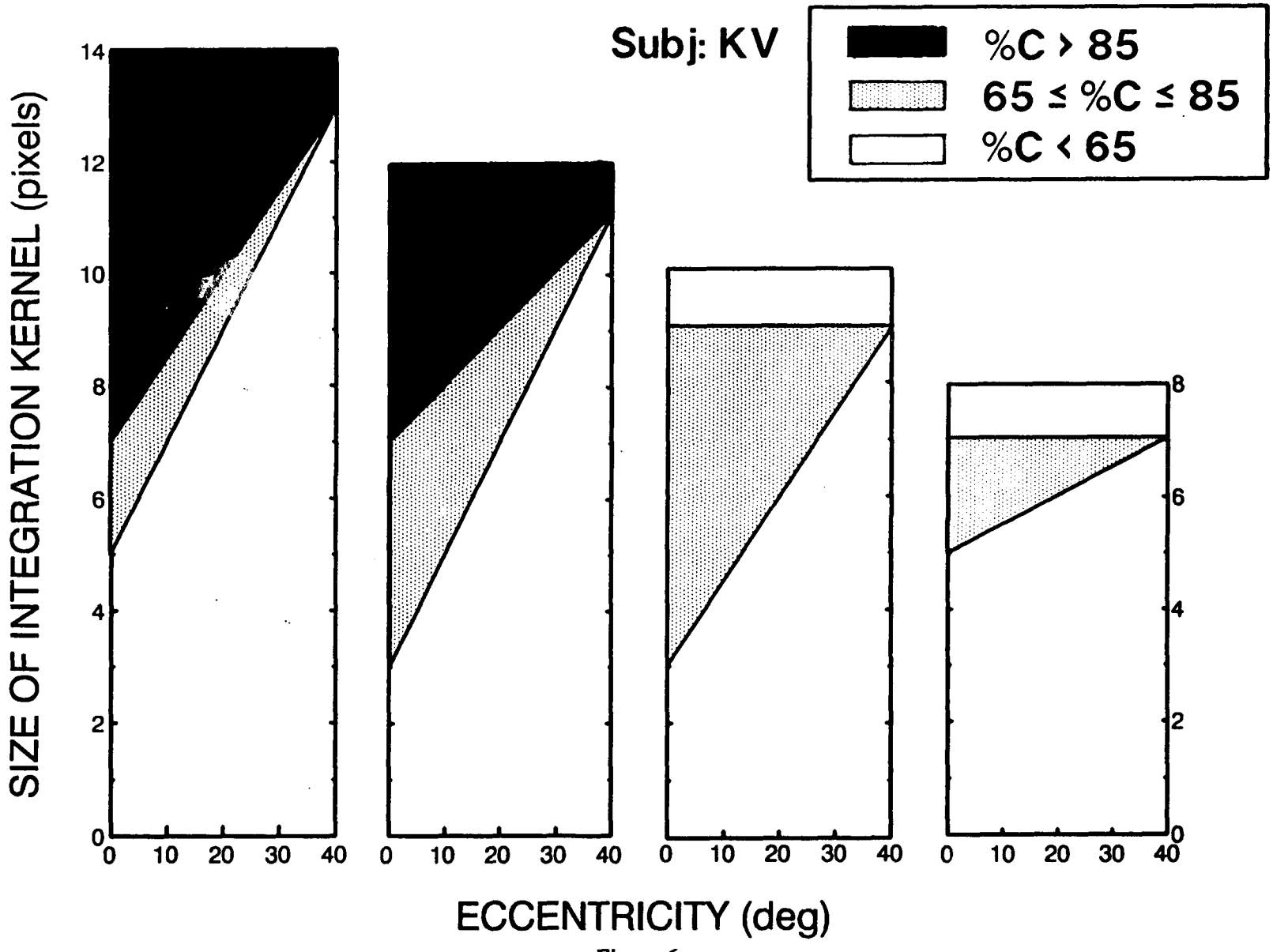


Figure 6c
Blur Discrimination Data for Observer KV

are taken together, this function is at or near threshold as defined in the present study. Of course, many of the other functions are also at or near threshold by that definition. It was not the purpose of the present study to estimate the CMFN, but rather to use the CMFN as an initial estimate of the threshold blur gradient for wide-field imagery.

The slope of the CMFN that produced a minimally discriminable image in the present study was very similar to the slopes estimated from luminance and contrast sensitivity data (Johnston, 1987; Rovamo, 1983; Rovamo, Virsu, & Näsänen,1978). This similarity at least suggests that the discrimination of blurred images may be dependent on image properties defined by such fundamental measures as component spatial frequency and orientation. Further, similar data have been obtained here for visually diverse, complex images, suggesting that the visual equivalence of variable-resolution images is independent of local variations in image detail.

The data of Figure 6 will be used below (see General Discussion) to estimate the minimal sampling rate required to adequately represent typical real-world imagery. In the context of image representation schemes recently reported (Ebrahimi & Kunt, 1991; Porat & Zeevi, 1988), these data may be used to generate efficient imagery that consists of only those spectral components to which each region of the visual field is sensitive (cf. Zeevi *et al.*, 1990).

EXPERIMENT 2: BLUR DISCRIMINATION AS A FUNCTION OF IMAGE CONFIGURATION AT 15° AND 30° ECCENTRICITY

INTRODUCTION

The data of Experiment 1 suggest that wide-field images processed using position-varying filters can appear preattentively similar to an unprocessed image for certain processing gradients. Furthermore, processing near the center of the image can be effectively exchanged for processing in the periphery in such a way as to maintain the perceptual equivalence of the image and its unprocessed counterpart. In Experiment 2, sensitivity to image blur was measured for circular and radial-segment apertures of various sizes. Data were obtained at 15° and 30° eccentricity in order to determine whether there are differences in the spatial properties of the visual mechanisms

underlying the detection of image blur at these two eccentricities.

METHOD

Observers

Data were obtained from three observers, one of whom (GG) participated in Experiment 1. Observers CT and DW were males, 20 and 26 years of age, respectively. Each had normal uncorrected vision. Observers CT and DW were unaware of the purpose of the study, and each was paid to participate in this experiment.

Stimuli and Apparatus

Three images were used as stimuli in this experiment. Two of the images were those used in Experiment 1, and the third was a black-and-white, one cycle/degree, radial square wave. The stimuli were chosen to represent three levels of spatial homogeneity (bombers, lowest; square wave, highest). The 1024 x 1024 x 8-bit images were processed by convolution (Equation 1) with gaussian kernels again varying in size from 3 x 3 to 13 x 13 pixels. The test stimuli were produced by applying either a circular or radial-segment aperture (see Figure 7) on both sides of the fixation point of both the original and processed images. The apertures were centered at either 15° or 30° along the horizontal meridian. The sets (circular and radial-segment) of stimuli presented at 15° eccentricity had areas of 8.4, 16.8, 25.2, 33.5, and 41.9 deg^2. The sets presented at 30° eccentricity had areas of 16.8, 33.5, 50.3, 67.0, and 83.8 deg^2 (i.e., corresponding to a magnification factor of two relative to the stimuli presented at 15°). The display system was identical to that used in Experiment 1.

Procedure

The testing procedure was identical to that of Experiment 1. In a given experimental session, all three stimuli and all five stimulus areas were tested (30 repetitions for each of the 15 conditions) for one eccentricity, one stimulus configuration, and one level of processing (i.e., blurring). Three randomized blocks of trials were run in each session, with the three stimulus types processed at one kernel size determining the block. For each of the three stimulus types, the five stimulus areas were tested in a random order as a sub-block. Each experimental session

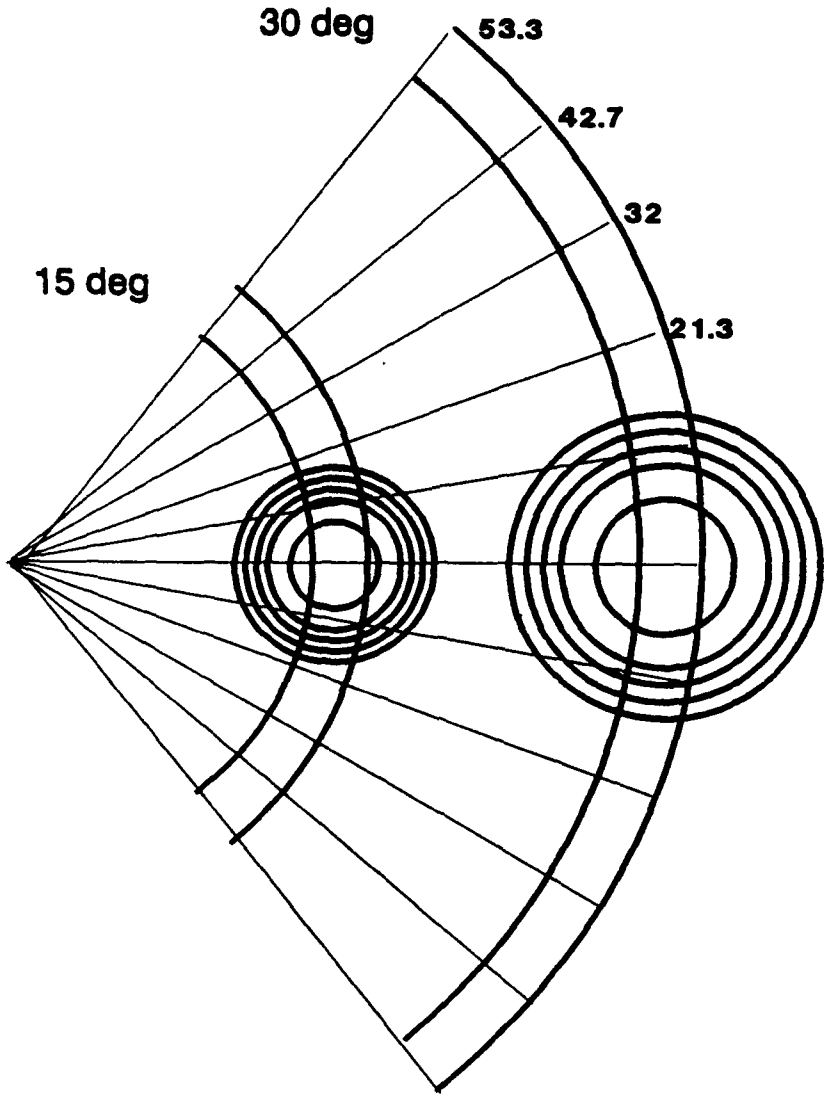


Figure 7
The Circular and Radial-Segment Apertures Used to
Define the Stimuli Used in Experiment 2

consisted of 450 trials, and required about 45 min to complete. For each observer, between three and five experimental sessions were run for each combination of eccentricity, stimulus configuration, and kernel size.

RESULTS

Shown in Figure 8 is a typical set of response functions used to estimate blur discrimination thresholds. The data of Figure 8 were obtained at 15° eccentricity for the lowest level of stimulus homogeneity (i.e., H1, the bombers) and represent an average over all three observers. The two sets of data in each panel correspond to either the circular (filled circles) or radial-segments stimuli, and the five panels correspond to the five stimulus areas tested. The data points in each panel show the proportion of correct responses as a function of the size of the summation kernel used to process the stimuli. As the summation kernel increases in size, the amount of blur increases and the proportion of trials for which the processed stimulus is discriminated from its unprocessed counterpart also increases. The functions placed through the data in each panel of Figure 8 represent the best fitting Weibull distribution of the form:

$$P = 1 - 0.5\exp\left[-\left(\frac{K}{K_t}\right)^s\right] \tag{12}$$

where P is the proportion of correct responses, K is the kernel size, K_t is the threshold kernel size (corresponding to P = 0.816), and s determines the steepness of the function. The function of Equation 12 was fitted to the data using the SigmaPlot Scientific Graph System (Jandel Scientific). The horizontal dotted line in the figure indicates a proportion correct of 0.816, which was taken as the threshold response level.

Blur thresholds, estimated from response functions like those of Figure 8, are plotted in Figure 9 as a function of stimulus area for both of the eccentricities tested (15° and 30°) and for all three levels of stimulus homogeneity (H1-H3). The functions obtained for both the circular and radial-segment stimuli show a general decrease in blur threshold as stimulus area is increased, although the decrease was generally greater for the data obtained at 15°. There appears also to be

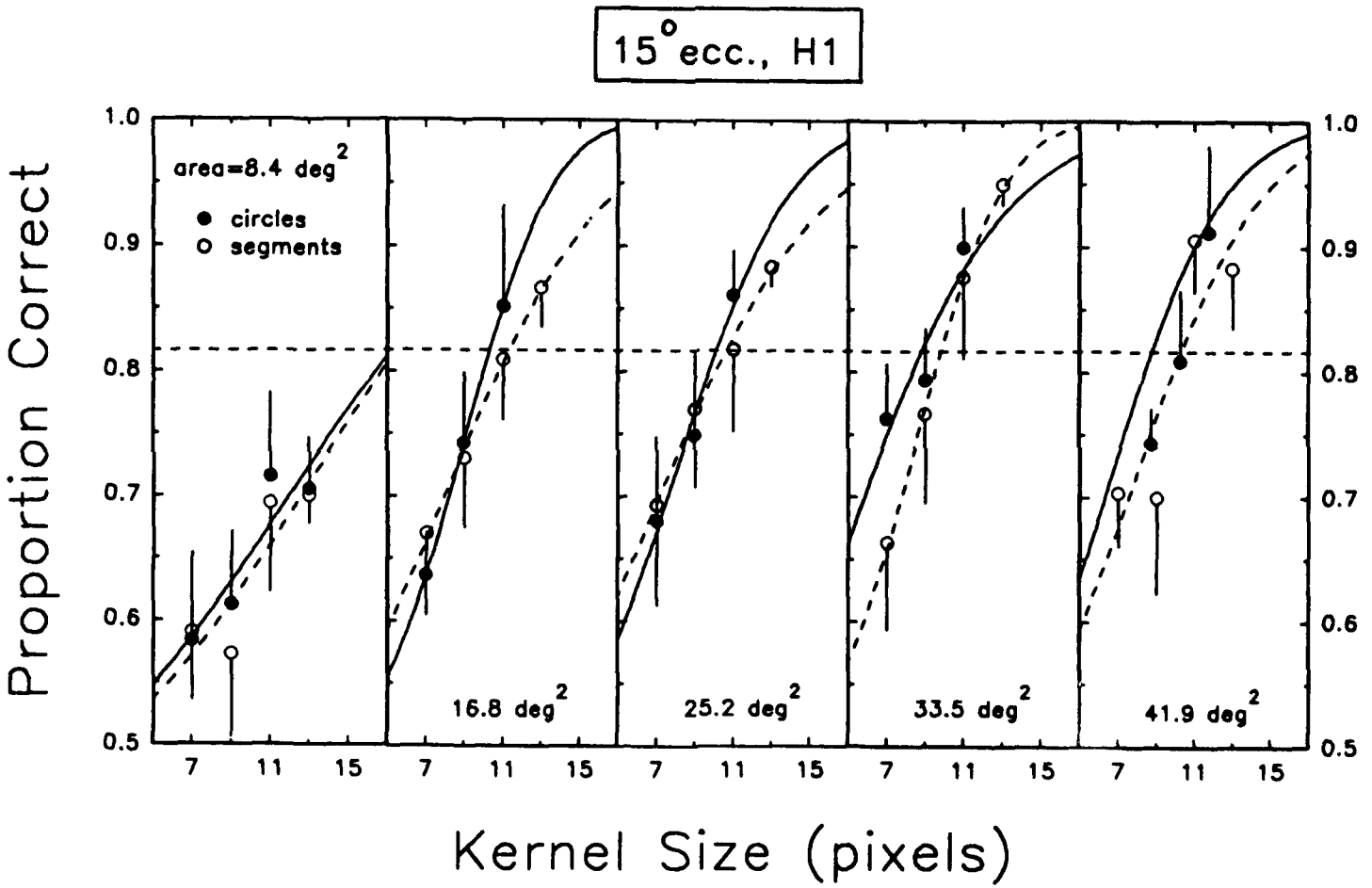


Figure 8
Response Functions Used to Estimate Blur Discrimination Thresholds

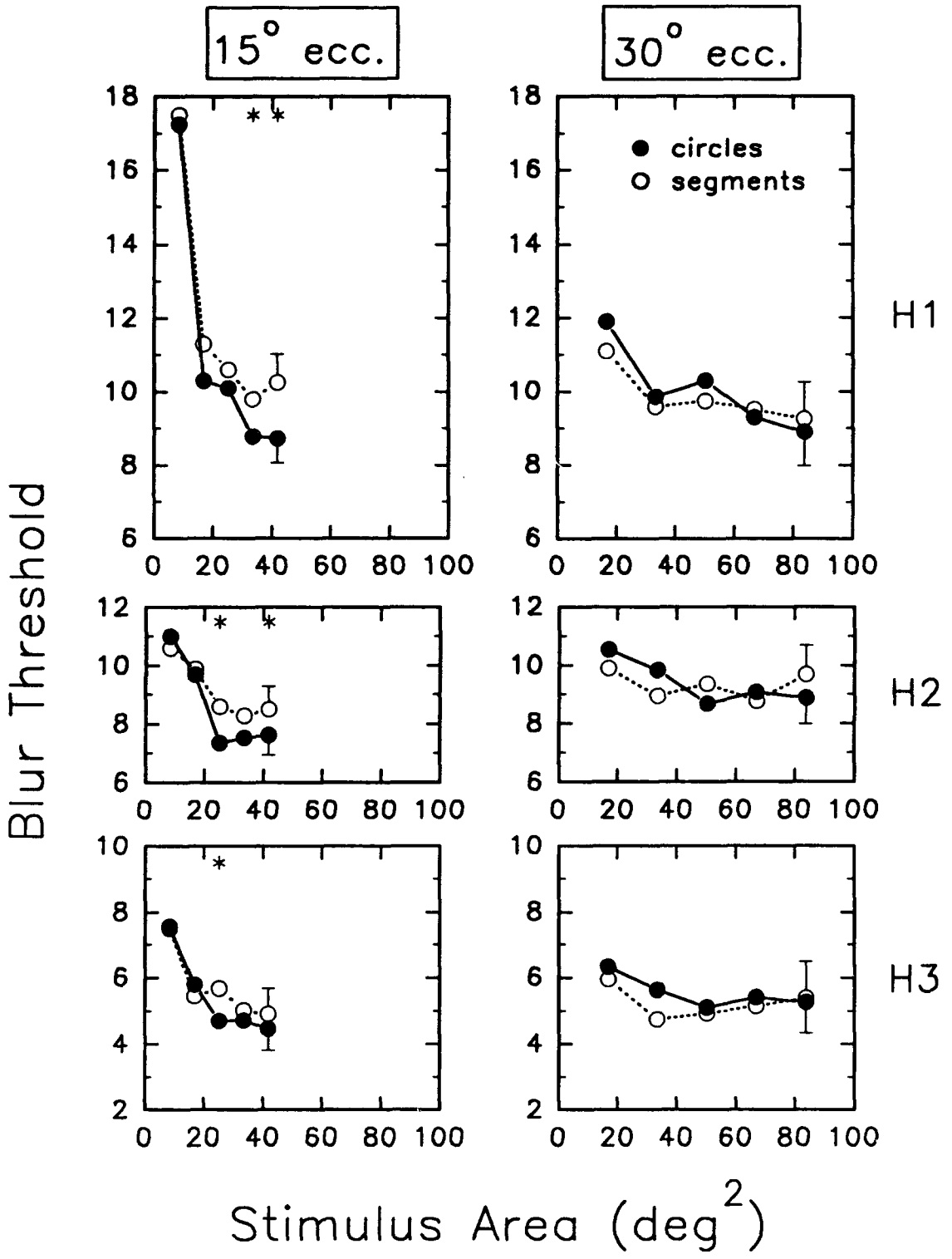


Figure 9
Blur Thresholds as a Function of Stimulus Area

a difference between the two eccentricities tested in the relative decrease in blur threshold for the two stimulus configurations. In order to test this difference, *t*-tests were performed on each of the differences between the two curves in each of the panels of Figure 9. Because five *t*-tests were performed for each eccentricity/homogeneity condition, the level of significance for the tests described here was reduced to p=0.01. The asterisks placed along the upper axis in each panel indicate the stimulus areas for which the blur thresholds for the circle and radial-segment stimuli were significantly different at this level. The computed *t*-values and degrees of freedom for each of the significant differences indicated by asterisks in the figure were as follows: $t_{df=18}$= 3.92, p<0.002 (H1/33.5); t_{18}= 5.28, p<0.001 (H1/41.9); t_{18}= 3.23, p<0.01 (H2/25.2); t_{20}= 3.06, p<0.01 (H2/41.9); and t_{18}= 3.16, p<0.01 (H3/25.2). As can be seen in Figure 7, the smallest circle and radial-segment stimuli are very similar in shape, and so little or no difference between the stimulus configurations would be expected for those areas. While only five of the thirty differences tested were statistically significant, all of the significant differences were found among the three largest areas at 15° eccentricity.

Shown in Figure 10 is the difference in blur threshold between the 15° and 30° eccentricity conditions for both the circle and radial-segment stimuli and for the only two stimulus areas (16.8 and 33.5 deg^2) tested at both eccentricities. For both stimulus areas, the difference in blur threshold between the two eccentricities tested was significantly different from zero for the circles (16.8:t_{18}= 1.00, p<0.01; 33.8:t_{16}= 4.04, p<0.002) but not for the radial segments (16.8:t_{18}= 0.42, p>0.50; 33.8:t_{18}= 0.14, p>0.80).

DISCUSSION

The data of Figure 9 show that there is a general decrease in blur discrimination threshold with increases in stimulus size. This is true for both the circular and segmental stimuli presented at both 15° and 30° eccentricity. There is also an indication in some of the functions that blur threshold has reached an asymptote and is even increasing for the largest stimulus areas. This effect suggests a functional limit to the size of the excitatory portion of the perceptual receptive field (Westheimer, 1967). The fact that the increase in threshold is more obvious for the segmental stimuli further suggests that the perceptual receptive field for these stimuli may be

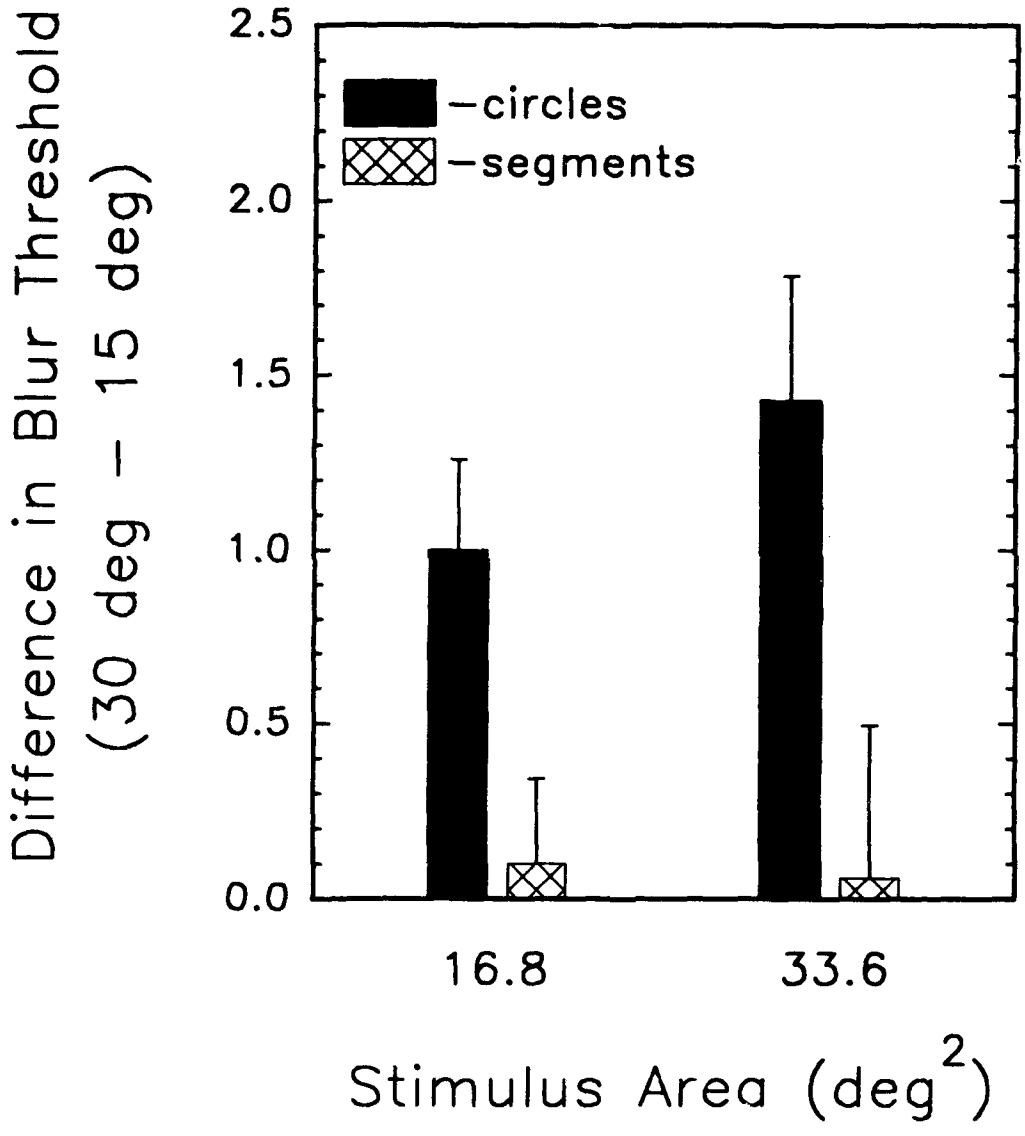


Figure 10
Difference in Blur Thresholds Between 15° and 30° Eccentricity

organized differently than for circular stimuli. These possibilities will be discussed further below.

The decrease in blur threshold with increases in stimulus area is generally greater for the stimuli presented at 15° suggesting that the effective magnification factor of two applied to the 30° stimuli did not fully compensate for the effect of stimulus area. It should be noted, however, that the difference in the threshold change is larger for the 15° stimuli due mainly to the relatively high threshold at the smallest stimulus size. Given that real-world stimuli were used for homogeneity levels H1 and H2, it is possible that this high threshold is related to the chance absence within the smallest aperture of those stimulus features that are most easily discriminated under the conditions of the present study. This interpretation is supported by the generally lower thresholds at all stimulus sizes for the square-wave stimuli, which were most spatially homogeneous. It is concluded, therefore, that a CMF of two adequately equates the blur discrimination data obtained at 15° and 30° eccentricity.

The difference in blur threshold obtained using circular and segmental stimuli is more obvious and only statistically significant at 15° eccentricity. Although some of the differences at 30° were relatively large, the variability at that eccentricity was also higher thus rendering the differences statistically insignificant. This is not an explanation for the relevant differences between 15° and 30° eccentricity, however, since the divergence at high stimulus areas, of the functions corresponding to the circular and segmental stimuli, is consistent only at 15° for the homogeneity levels tested. Thus, the data of Figure 9 suggest that at 15° eccentricity (and presumably at lesser eccentricities), the visual information required to discriminate image blur is not summated within a constant eccentricity. Rather, the perceptual summation area is more nearly circularly symmetrical. On the other hand, perceptual summation at 30° extends over a relatively large area and is generally unaffected by the spatial distribution of the stimulus.

As noted earlier, two of the areas tested at 15° eccentricity correspond to areas tested also at 30° eccentricity. As can be seen in the data of Figure 10, the difference in blur threshold between 15° and 30° eccentricity for these two areas is much larger for the circular stimuli than for the segmental stimuli. It was concluded above, based on the data from all stimulus areas tested, that a CMF of two was sufficient to equate the blur discrimination data obtained at 15° and 30°. That conclusion is consistent only with the circular data of Figure 10. There appears, however, to

be no significant difference in the blur thresholds obtained at the two eccentricities when segmental stimuli are used. This is further evidence of a difference in the spatial properties of perceptual fields at different eccentricities.

GENERAL DISCUSSION

In the case of the variable-resolution images used in the present study, the distortion function is derived from physiological and psychophysical data that describe how visual processing capability decreases with eccentricity. Since the distortion function is known, there are two ways of representing the variable-resolution image by a discrete set of samples without generating artifacts. First, the image can be transformed such that it can be properly represented by a uniform set of samples. This is done by using the inverse of the distortion function to produce a bandlimited image (without loss of information) and then sampling the image uniformly in accordance with the standard Nyquist criterion (Jerri, 1977). Stated another way, if an image can be bandlimited by a transformation which appropriately distorts the spatial axis, then it can be adequately represented by a discrete set of samples. The second way of representing a variable-resolution image by a discrete set of samples is to sample it nonuniformly at a rate determined by the form of the distortion function. The image can then be reconstructed using interpolation functions (i.e., sinc functions distorted in accordance with the nonuniform spatial distortion function). Thus, in this case, the variable-resolution image can be represented by a finite set of sampling points even though it is not bandlimited and as such does not satisfy the Nyquist condition.

In the context of the present study and the applications discussed earlier, we start from a uniform-resolution image and generate a variable-resolution image. The operation performed in this case can be viewed as filtering which is analogous to the bandpass filter applied prior to uniformly sampling an image. The low-pass filtering is performed in our case by convolving the image with a gaussian. This operation does not satisfy the requirement of projecting an image onto the space of BLFs in that the associated measure is an effective bandwidth (Gabor, 1946) and hence is only approximate. [For examples of variable-resolution images generated by filtering with a nonuniform sinc function, see Zeevi *et al.* (1988) and Peterfreund and Zeevi (1992)]. In

the case of nonuniform sampling, the filter applied prior to sampling is nonuniform. In other words, its properties (corresponding bandwidth) vary as a function of position in accordance with visual requirements.

Efficient Sampling of Variable-Resolution Imagery

Numerous image processing and compression techniques have been developed for representing visual images efficiently. These are, in general, based on the redundancy in natural images and the 2-D spatial frequency selectivity of the human visual system (Baddeley & Hancock, 1991; Field, 1987; Reed, Ebrahimi, Marqués, & Kunt, 1991). Variable-resolution imagery takes advantage of another salient property of the human visual system--its spatial inhomogeneity. One practical advantage of variable-resolution imagery, as defined here (see Equation 2), is that it can be adequately (by a visual criterion) sampled by fewer points than its fixed-resolution counterparts. The variable-resolution technique described here was developed to efficiently represent images used in high-fidelity, wide field-of-view displays (Geri, Zeevi, & Porat, 1990). Thus, for the purposes of image storage, manipulation, and transmission, it is desired to exploit the variable-resolution properties of the image such that the data set can be reduced accordingly. The question then is, how should the sampling points be *nonuniformly* distributed over the image field such that they adequately represent the image in the sense that the variable-resolution image can be reconstructed from this set of sampling points? Intuitively, it is clear that as the data become more sparsely distributed as a result of variable-resolution processing, the required density of sampling points decreases. In theory, the sampling rate appropriate for a given variable-resolution image is determined by the local bandwidth as discussed above. In practice, we do not have an exact measure of the local bandwidth, therefore we have chosen to use the distortion function to obtain the appropriate sampling rate for our variable-resolution images. As noted by Zeevi & Shlomot (1993), the local bandwidth is related to the derivative of the distortion function. According to the sampling theorem for images which belong to the space of LBLFs (Zeevi & Shlomot, 1993), the set of nonuniformly distributed sampling points can be obtained by applying the inverse of the distortion function to the uniform grid. This has been done in one dimension using as a distortion function one of the functions of Figure 3 that was found to be at or near threshold for all of the observers tested in Experiment 1.

Shown on the left in Figure 11 is the chosen distortion function. Shown at the top right in the same figure is a set of uniformly-spaced sampling points (i.e., pixels) corresponding to one-half (from center to edge) of a cross-cut through an image. Shown at the bottom right in Figure 11 is the nonuniform set of sampling points that corresponds to the chosen distortion function. In this case, the operation results in a reduction by a factor of about two in the number of points required for the representation. This would correspond to a reduction of about four when the full two-dimensional image is represented at the indicated sampling rate. It should be noted that this is a conservative estimate of the savings that might be expected from variable-resolution processing of *static* imagery in that the psychophysical technique used here for assessing the perceptual equivalence of the processed and unprocessed images was designed to detect very small differences in the appearance of the images. Much larger differences in the images might be tolerated if the perceptual criterion was recognition or identification, and the associated sampling efficiency in that case would be significantly greater.

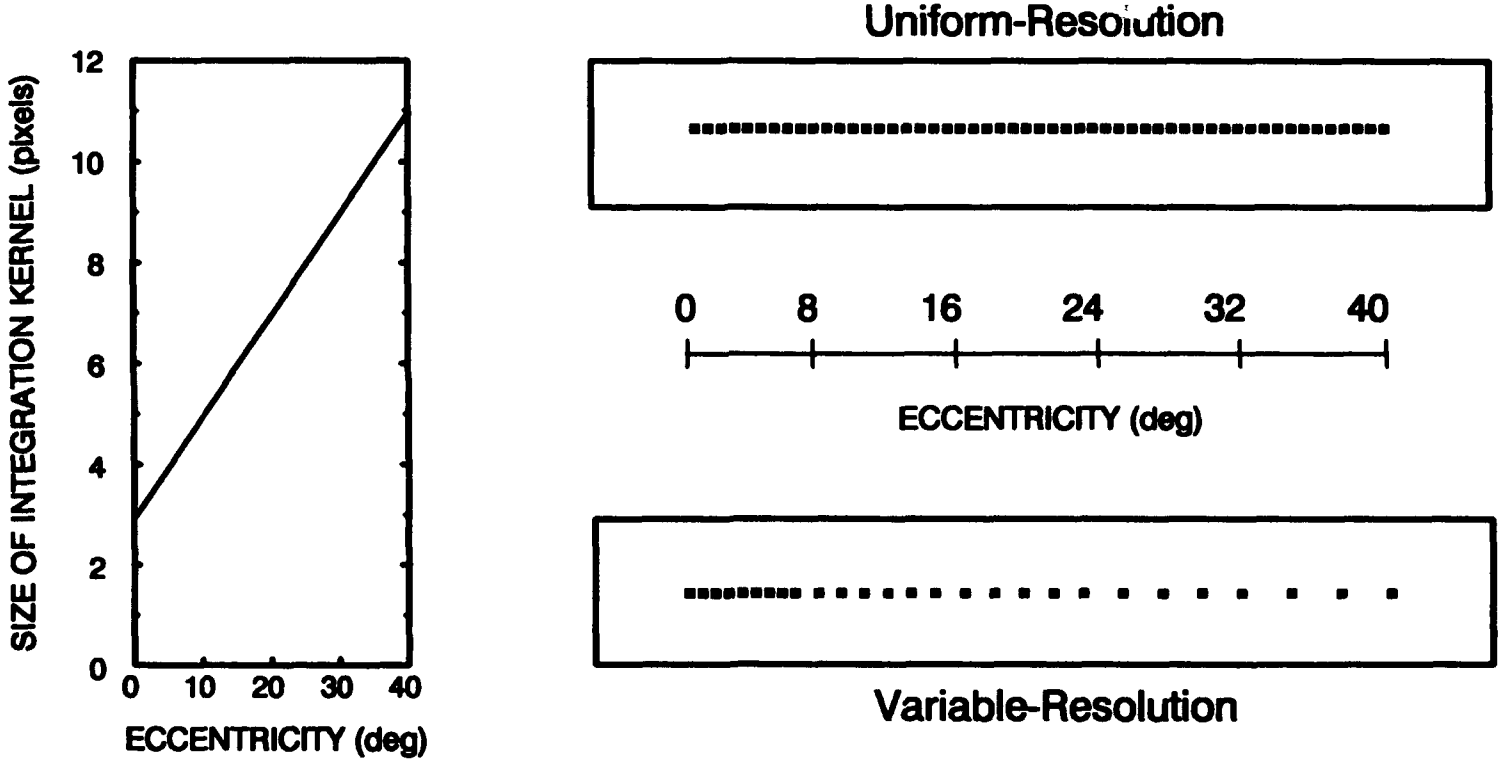


Figure 11
Nonuniform Sampling According to a Measured Distortion Function

REFERENCES

Aronszajn, N. (1950). Theory of reproducing kernels. *Transactions of the American Mathematical Society*, **68**, 337-404.

Baddeley, R.J. & Hancock, P.J.B. (1991). A statistical analysis of natural images matches psychophysically derived orientation tuning curves. *Proceedings of the Royal Society of London* B, **246**, 219-223.

Clark, J.J., Palmer, M.R., & Lawrence, P.D. (1985). A transformation method for the reconstruction of functions from nonuniformly spaced samples. *IEEE Transactions in Acoustics, Speech, & Signal Processing*, **33**, 1151-1165.

Daniel, P.M. & Whitteridge, D. (1961). The representation of the visual field in the cerebral cortex in monkeys. *Journal of Physiology (London)*, **159**, 203-221.

Dow, B.M., Snyder, A.Z., Vautin, R.G., & Bauer, R. (1981). Magnification factor and receptive field size in foveal striate cortex of the monkey. *Experimental Brain Research*, **44**, 213-228.

Ebrahimi, T. & Kunt, M. (1991). Image compression by Gabor expansion. *Optical Engineering*, 30, 873-880.

Field, D.J. (1987). Relations between the statistics of natural images and the response properties of cortical cells. *Journal of the Optical Society of America* A, **4**, 2379-2394.

Gabor, D. (1946). A theory of communication. *Journal of the I.E.E.*, **93**, 429-459.

Geri, G.A., Zeevi, Y.Y., & Porat, M. (1990). Efficient image generation using localized frequency components matched to human vision, (*AFHRL-TR-90-25, AD A224 903*). Williams Air Force Base AZ: Operations Training Division, Air Force Human Resources Laboratory.

Horiuchi, H. (1968). Sampling principle for continuous signals with time-varying bands. *Information and Control* **13**, 53-61.

Jerri, A. (1977). Shannon sampling theorem--Its various extensions and applications: A tutorial review. *Proceedings of the I.E.E.E.* **65**, 1565-1596.

Johnston, A. (1987). Spatial scaling of central and peripheral contrast-sensitivity functions. *Journal of the Optical Society of America* A, **4**, 1583-1593.

Kelly, G.R. (1992). Measurement of modulation transfer functions of simulator displays. (*AL-TP-1992-0056*). Williams Air Force Base AZ: Aircrew Training Research Division, Human Resources Directorate, Armstrong Laboratory.

Levi, D.M., Klein, S.A., & Aitsebaomo, A.P. (1985). Vernier acuity, crowding and cortical magnification. *Vision Research* **25**, 963-977.

Peterfreund, N. & Zeevi, Y.Y. (1990). Image representation in nonuniform systems, In V. Cantoni, L.P. Cordella, S. Levialdi, & G. Sanniti di Baja,(Eds), *Progress in image analysis and processing*, (pp. 199-203),Singapore:World Scientific.

Peterfreund, N. & Zeevi, Y.Y. (1992). Nonuniform image representation in area-of-interest systems. *E.E. Publication 866*, Technion, Haifa, Israel.

Porat, M. & Zeevi, Y.Y. (1988). The generalized Gabor scheme of image representation in biological and machine vision. *I.E.E.E. Transactions in Pattern Analysis and Machine Intelligence*, **10**, 452-468.

Reed, T.R., Ebrahimi, T., Marqués, F., & Kunt, M. (1991). New generation methods for the high-compression coding of digital image sequences, In H. Burkhardt, Y. Neuvo, J.C. Simon, (Eds), *From pixels to features II*, (pp. 401-413),North-Holland:Elsevier.

Rovamo, J. (1983). Cortical magnification factor and contrast sensitivity to luminance-modulated chromatic gratings. *Acta Physiologica Scandanavia*, **119**, 365-371.

Rovamo, J. & Raninen, A. (1984). Critical flicker frequency and M-scaling of stimulus size and retinal illumination. *Vision Research*, **24**, 1127-1131.

Rovamo, J. & Virsu, V. (1979). An estimation and application of the human cortical magnification factor. *Experimental Brain Research*, **37**, 495-510.

Rovamo, J., Virsu, V. & Näsänen, R. (1978). Cortical magnification factor predicts the photopic contrast sensitivity of human vision. *Nature*, **271**, 54-56 (1978).

Talbot, S.A. & Marshall, W.H. (1941). Physiological studies on neural mechanisms of visual localization and discrimination. *American Journal of Ophthalmology*, **24**, 1255-1264.

Van Essen, D.C., Newsome, W.T. & Maunsell, H.R. (1984). The visual field representation in striate cortex of the macaque monkey: Asymmetries, anisotropies, and individual variability. *Vision Research*, **24**, 429-448.

Virsu, V., Näsänen, R., & Osmoviita, K. (1987). Cortical magnification and peripheral vision. *Journal of the Optical Society of America* A, **4**, 1568-1578.

Westheimer, G. (1967). Spatial interaction in human cone vision. *Journal of Physiology, London*, 190, 139-154.

Zadeh, L.A. (1952). A general theory of linear signal transmission systems. *Journal of the Franklin Institute*, **253**, 293-312.

Zeevi, Y.Y., Peterfreund, N., & Shlomot, E. (1988). Pyramidal image representation in nonuniform systems. *S.P.I.E. Proceedings in Visual Communications & Image Processing*, **1001**, 563-571.

Zeevi, Y.Y., Porat, M., and Geri, G.A. (1990). Computer image generation for flight simulators: the Gabor approach. *The Visual Computer*, **6**, 93-105.

Zeevi Y.Y. & Shlomot, E. (1993). Nonuniform sampling and anti-aliasing in image representation. *I.E.E.E. Transactions on Signal Processing*, **41**, 1223-1236.

APPENDIX A

The program, *vresfb.c*, used to generate variable-resolution imagery.

```
C               Filename:                       vresfb.f
C               Created By:                     Craig A. Vrana - UDRI
C               Creation Date:          27-MAR-90
C               Modified By:            Craig A. Vrana - UDRI
C               Last Modified:          06-MAY-92
C               Language:                       FORTRAN-77
C
C
C       This program reads an array of pixels (which must be square),
C representing an image in cartesian coordinates, and generates an
C output image with variable resolution.
C
C       Each output pixel represents the average of K x K pixels where K is the
C dimension of the "receptive field" over which the original image is
C locally averaged.
C
C       The size of the receptive field, as a function of eccentricity (R), is
C determined by; K=B*R+C, where B and C are constants determined by the
C particular cortical magnification function chosen.
C
C       The input image filename is of the form xxxxxxx.IMG, where the user
C enters the xxxxxxx (7 characters) when asked for the input image filename.

C                               Modified:       05-DEC-90
C                                       Code was added to allow the processing of binary image files
C                                       instead of the large ASCII files as originally coded.
C                                       This program must now be linked with iotools.o


        PROGRAM variable_resolution


C ***************************************************************************
C                                               CONSTANT DEFINITIONS
C ***************************************************************************

C       Define the device number used for keyboard input
        INTEGER         KEYBOARD
        PARAMETER       (KEYBOARD = 5)

C       Define the maximum size for the image header
        INTEGER         MAX_HEADER_SIZE
        PARAMETER       (MAX_HEADER_SIZE = 316)

C       Define the maximum size of the image, in pixels, in one dimension
        INTEGER         MAX_IMAGE_SIZE
        PARAMETER       (MAX_IMAGE_SIZE = 1024)

C       Define the maximum size of the weighting kernal
        INTEGER         MAX_GAUSS_KERNEL
        PARAMETER       (MAX_GAUSS_KERNEL = 75)

C       Define the maximum number of chars in the filename of the image
        INTEGER         MAX_NAME_SIZE
        PARAMETER       (MAX_NAME_SIZE = 64)

C       Define the value to be used for PI.
        REAL*8          PI
```

FILE=a:vresfb.f * PAGE=1

```
120:                 mode(1:) = 'r'
122:                 mode(2:) = ' '
123:                 mode(3:) = CHAR(0)
124:                 in_handle = open_bfile(image_name, mode, error_code)
125:                 WRITE (SCREEN,*) error_code
126:                 WRITE (SCREEN,*) in_handle
127:                 IF(error_code .NE. 0) THEN
128:                       GOTO 1000
129:                 ENDIF
130:
131:
132: C       WRITE   (SCREEN,*)      'Enter image size : '
133: C       READ    (KEYBOARD,*)    size
134:         size = 1024
135:         half_size = size / 2
136:
137: C       OPEN(UNIT=10, FILE=image_name, ACCESS='SEQUENTIAL', ERR=1000, STATUS='OLD', READONLY)
138:
139:
140: C       Get the name of the output file
141:         WRITE (SCREEN,*)        'Enter the name of the output file : '
142:         READ  (KEYBOARD,10)     file_name
143:
144:         dummy = index(file_name, ' ')
145:         file_name(dummy:) = CHAR(0)
146:
147: C       OPEN(UNIT=20, FILE=file_name, ACCESS='SEQUENTIAL', ERR=4000, STATUS='UNKNOWN')
148:
149:                 mode(1:) = 'w'
150:                 mode(2:) = ' '
151:                 mode(3:) = CHAR(0)
152:                 out_handle = open_bfile(file_name, mode, error_code)
153:                 WRITE (SCREEN,*) error_code
154:                 WRITE (SCREEN,*) out_handle
155:                 IF(error_code .NE. 0) THEN
156:                       GOTO 4000
157:                 ENDIF
158:
159:
160: C       Get cmf constants from user
161:  25     WRITE (SCREEN,*) 'Enter kernel constants (i.e., k1 <space> k2 <ret>)'
162:                 READ  (KEYBOARD,*) k1, k2
163:
164:
165:
166: C   Get start time
167:         CALL TIME(current_time)
168:
169:                 open(15, FILE = 'vresf.knl')
170:  98     FORMAT(1X, 'x= ', I3, 3X, 'y= ', I3, 3X, F5.3)
171: C  use a single kernel
172:         kernel = 1
173:
174:         IF (k1 .GT. k2) THEN
175:                 k = k1
176:         ELSE
177:                 k = k2
178:         ENDIF
179:

FILE=a:vresfb.f  *  PAGE=3
```

```
180:         WRITE   (SCREEN,*)      '===> Calculating weighting kernels'
182:                 DO 100 x=0, k
183:                         DO 100 y=0, k
184:                                 radius = (DBLE(x)/DBLE(k1))**2 + (DBLE(y)/DBLE(k2))**2
185:                                 IF ((kernel .EQ. 0) .OR. (radius .EQ. 0)) THEN
186:                                         w = 1.0000
187:                                 ELSE
188:                                         w = exp(-(4*radius))
189:                                 ENDIF
190:                                 weight(y,x,kernel) = w
191:                                 write(15,98) x, y, w
192:  100    CONTINUE
193:
194:                 close(15)
195:
196:         WRITE   (SCREEN,*)      '===> Reading image'
197:
198:         num_bytes = size*size
199:
200:         WRITE(SCREEN,*)num_bytes, 'bytes to be read'
201:
202: C       READ(10,*)(c_image(d),d=0,num_bytes-1)
203:
204:                 dummy1 = read_bfile(in_handle, num_bytes, c_image(0))
205:
206:                 WRITE (SCREEN,*) 'Number of bytes read = ', dummy1
207:                 IF (dummy1 .NE. num_bytes) THEN
208:                         GOTO 2000
209:                 ENDIF
210:
211: C       Close file
212:
213: C       CLOSE(10, ERR=2010, STATUS='KEEP')
214:
215:                 error_code = close_bfile(in_handle)
216:                 WRITE (SCREEN,*) error_code
217:                 IF (error_code .NE. 0) THEN
218:                         GOTO 2010
219:                 ENDIF
220:
221:
222:         WRITE   (SCREEN,*)      '===> Converting to Cartesian coordinates'
223:         DO 200 x = 0, size-1
224:                 DO 200 y = 0, size-1
225:                         image(y,x) = AND(INT(c_image(y+x*size)), 255)
226:  200  CONTINUE
227:
228:
229: C       This routine calculates the variable resolution image and returns it
230: C in the out array.
231:
232:
233: C       Do calculations
234: C       Get radial delta value for 40 degrees maximum  (degrees per pixel)
235:         dpp = 40.0 / half_size
236:         WRITE (SCREEN, *) '===> Filtering image'
237:         DO 330 x = 0, size-1
238:                 WRITE (SCREEN,*) 'x=', x
239:            DO 330 y = 0, size-1

FILE=a:vresfb.f  *  PAGE=4
```

```
                        t    = 0.0
                        wsum = 0.0
                        DO 320 m = -k, k
                            absm = IABS(m)
                            DO 320 n = -k, k
                                absn = IABS(n)
                                w = weight(absn,absm,kernel)
C                               MIN ensures we don't go past end of image
                                ypn = MIN (y + n, size-1)
                                ypn = MAX (ypn, 0)
                                xpm = MIN (x + m, size-1)
                                xpm = MAX (xpm, 0)
                                wsum = wsum + w
C                               t sums pixel values in each subarea of image
 310                    t = (image(ypn, xpm)*w) + t
 320            CONTINUE
                                IF (wsum .LT. 0.0001) THEN
                                    WRITE(SCREEN,*) 'wsum = 0  x = ',x,'y = ',y
                                ELSE
                        out(y, x) = NINT(t/wsum)
                                ENDIF
 330    CONTINUE


        ma = out(10,10)
        mi = ma
        DO 350 x = 0, size-1
                DO 350 y = 0, size-1
                        IF (out(y,x) .GT. ma) THEN
                                ma = out(y,x)
                        ELSE IF (out(y,x) .LT. mi) THEN
                                mi = out(y,x)
                        ENDIF
 350    CONTINUE


        WRITE(SCREEN,*) '===> Scaling Image', mi, ma

                WRITE (SCREEN,*) '===> Converting to linear array.'
        DO 400 x = 0, size-1
                DO 400 y = 0, size-1
                        c_image(y+x*size) = out(y,x)
 400    CONTINUE

        WRITE (SCREEN,*) '===> Writing image'
C       WRITE(20,500) (c_image(d),d=0,num_bytes-1)
C 500   FORMAT(16(1X,I3.3))

                dummy1 = write_bfile(out_handle, num_bytes, c_image(0))

                WRITE (SCREEN,*) 'Number of bytes written = ', dummy1
                IF (dummy1 .NE. num_bytes) THEN
                        GOTO 4010
                ENDIF

C       Close file
C       CLOSE(20, ERR=4030, STATUS='KEEP')


FILE=a:vresfb.f  *  PAGE=5
```

```
300:                error_code = close_bfile(out_handle)
302:                WRITE (SCREEN,*) error_code
303:                IF (error_code .NE. 0) THEN
304:                        GOTO 4030
305:                ENDIF
306:
307:
308: C       Print program timing and end program
309:         WRITE (SCREEN,*)        'Start time  =>', current_time
310:         CALL TIME (current_time)
311:         WRITE (SCREEN,*) 'Finish time =>', current_time
312:         WRITE (SCREEN,*) 'Program terminated normally.'
313:         STOP
314:
315:
316:
317: C ***************************************************************************
318: C                                                              EXCEPTIONS
319: C ***************************************************************************
320:
321: 1000    WRITE (SCREEN,*) 'Could not open ',image_name, '\b for reading.'
322:                 STOP
323:
324: 2000    WRITE (SCREEN,*)        'Error reading from ', image_name, '\b.'
325:                 WRITE (SCREEN,'(A13A)') ' UNIX error code =', error_code, '.'
326:                 STOP
327:
328: 2010    WRITE (SCREEN,*)        'Error closing ', image_name, '\b.'
329:                 STOP
330:
331: 4000    WRITE (SCREEN,*)        'Could not open',       file_name, '\bfor writing.'
332:                 WRITE (SCREEN,'(A13A)') ' UNIX error code =', error_code, '.'
333:                 STOP
334:
335: 4010    WRITE (SCREEN,*)        'Error writing to', file_name, '\b.'
336:                 WRITE (SCREEN,'(A13A)') ' UNIX error code =', error_code, '.'
337:                 STOP
338:
339: 4030    WRITE (SCREEN,*)        'Error closing', file_name, '\b.'
340:                 WRITE (SCREEN,'(A13A)') ' UNIX error code =', error_code, '.'
341:                 STOP
342:
343:                 END
344:
```

APPENDIX B

The programs used in the experimental portion of Experiments 1 and 2. Programs *opaste.c*, *slice.c*, and *vcirc.c* were used to generate the experimental stimuli from the unprocessed original images and the associated variable-resolution images. Program *scale.c* was used to apply the luminance gamma-correction as well as the correction for luminance variations across the image caused by projector optics and the directional properties of the rear-projection screen. Program *segs.c* was used to present the experimental stimuli and collect the observers' responses in Experiment 2, and was similar to the program used for this purpose in Experiment 1. Finally program *asegs.c* was used to analyze the observers' responses.

```
/*****************************************************************************
*                                                                            *
*                    Program:          opaste.c                                *
*                    Created by:     Craig A. Vrana                         *
*                    Creation date:  25-SEP-89                              *
*                    Last Modified:  06-DEC-90                              *
*                                                                            *
*****************************************************************************/
/*
                    Modified:   06-DEC-90
                        fixed problem of entering the default gap size and
                        having the program continue waiting fro input of a
                        gap size.

                    Modified:   16-NOV-90
                        Added the gap size to the command-line argument list
                        in order to allow the user to controll the gap-size
                        when executing this program.
*/


# include <stdio.h>
# include <string.h>
# include <ctype.h>

# define BACKGROUND          104        /* value to set gap to                  */
# define FILENAME_SIZE       64      /* max number of chars in filename     */
# define IMAGE_EXTENSION     ".img"  /* default image extension             */
# define MAX_IMAGE_SIZE      1024    /* max number of pixels in one line    */
# define GAP                 60      /* The number of pixels between images */

typedef unsigned char   byte;
typedef unsigned int    word;


void main(int argc,char *argv[])
{
    char    input_left[FILENAME_SIZE],
            input_right[FILENAME_SIZE],
            output[FILENAME_SIZE],
            side_to_use[2],
            gap_size[5];

    byte    buffer1[MAX_IMAGE_SIZE],
            buffer2[MAX_IMAGE_SIZE],
            out_line_buf[MAX_IMAGE_SIZE],
            *p1,
            *p2,
            *p3;

    word    gap,
            in_size1,
            in_size2,
            out_size,
            i,
            i1,
            i2,
            j;


FILE=a:opaste.c  *  PAGE=1
```

```
    FILE    *in1,
              *in2,
              *out;

    fprintf(stdout,"argc = %d\n", argc);

    if(argc != 6)
    {
        /*  get left input filename  */
        printf ("Enter left side input filename (.IMG):  ");
        scanf ("%s", input_left);

        /*  get right input filename  */
        printf ("Enter right side input filename (.IMG):  ");
        scanf ("%s", input_right);

        /*  get output filename  */
        printf ("Enter output filename (.IMG): ");
        scanf ("%s", output);

        /* get which half of each image to use  */
        printf ("Enter r or l for the half of the images to paste to gether: ");
        scanf ("%s", side_to_use);
        side_to_use[0] = toupper(side_to_use[0]);
        printf("Using %s side of images.\n", side_to_use);

        /* get the gap size in pixels  */
        printf ("Enter gap size in pixels: <%d>", GAP);
        if(scanf ("%d", &gap) != 1)
        {
            printf("Did not receive a gap size.\n");
            gap = GAP;
        }
        printf("Using %d pixel gap between images.\n", gap);

    }

    else
    {
        strcpy(input_left, argv[1]);
        strcpy(input_right, argv[2]);
        strcpy(output, argv[3]);
        strcpy(side_to_use, argv[4]);
        side_to_use[0] = toupper(side_to_use[0]);
        strcpy(gap_size, argv[5]);
        gap = ((int)atoi(gap_size)/2)*2;
        printf("Using %d pixel gap between images.\n", gap);
    }

    /*  append .img if necessary  */
    if (strrchr(input_left, '.')==NULL)
    {
        strcat (input_left, IMAGE_EXTENSION);
    }

    /*  open input file and setup buffer  */
    if ((in1=fopen(input_left, "r")) == NULL)
    {
        fprintf (stderr, "Could not open %s for reading.\n", input_left);
```

FILE=a:opaste.c * PAGE=2

```
120:         exit (1);
122:     }
123:
124:     /*  append .img if necessary  */
125:     if (strrchr(input_right, '.')==NULL)
126:     {
127:         strcat (input_right, IMAGE_EXTENSION);
128:     }
129:
130:     /*  open input file and setup buffer  */
131:     if ((in2=fopen(input_right, "r")) == NULL)
132:     {
133:         fprintf (stderr, "Could not open %s for reading.\n", input_right);
134:         exit (1);
135:     }
136:
137:
138:     /*  append .img if necessary  */
139:     if (strrchr(output, '.')==NULL)
140:         strcat (output, IMAGE_EXTENSION);
141:
142:     /*  open output file  */
143:     if ((out=fopen(output, "w")) == NULL)
144:     {
145:         fprintf (stderr, "Could not open %s for writing.\n", output);
146:         exit (1);
147:     }
148:
149:
150:     in_size1 = 1024; /* Change here for larger images */
151:     in_size2 = 1024; /* Change here for larger images */
152:
153:     if(in_size1 == in_size2)
154:     {
155:         out_size = in_size1;
156:     }
157:
158:     else
159:     {
160:         printf("\nThe two input images are different sizes.\n");
161:
162:         /*  close files  */
163:         fclose(in1);
164:         fclose(in2);
165:         fclose(out);
166:         exit(-1);
167:     }
168:
169:
170:     while (side_to_use[0] == 'L')  /* Paste left halves of images together*/
171:     {
172:         /*  read in one input line  */
173:         if (fread (buffer1, sizeof(byte), in_size1, in1) != in_size1)
174:             break;
175:         if (fread (buffer2, sizeof(byte), in_size2, in2) != in_size2)
176:             break;
177:
178:         p1  = buffer1;
179:         p2 = buffer2 + in_size2/2;
```

```
180:
182:
183:        /* The following loop reads in the first half the output image from
184:            the left side image (pointed to by p1) and the second half of the
185:            right side image (pointed to by p2) and combines them in the output
186:            buffer line by line. */
187:
188:        while (p1 < buffer1+in_size1/2)
189:        {
190:
191:            if(p1-buffer1 <= in_size1/2-gap/2)
192:            {
193:                out_line_buf[p1-buffer1]    = *p1;
194:            }
195:            else
196:            {
197:                out_line_buf[p1-buffer1] = BACKGROUND;
198:            }
199:
200:            if(p1-buffer1 >= gap/2)
201:            {
202:                out_line_buf[(p1-buffer1)+in_size1/2]  = *p2;
203:            }
204:            else
205:            {
206:                out_line_buf[(p1-buffer1)+in_size1/2]  = BACKGROUND;
207:            }
208:
209:            p1++;
210:            p2--;
211:        }
212:
213:        fwrite(out_line_buf, sizeof(byte), out_size, out);
214:    }   /*  for each line in the input file  */
215:
216:    while (side_to_use[0] == 'R')  /* Paste right halves of images together*/
217:    {
218:        /*  read in one input line  */
219:        if (fread (buffer1, sizeof(byte), in_size1, in1) != in_size1)
220:            break;
221:        if (fread (buffer2, sizeof(byte), in_size2, in2) != in_size2)
222:            break;
223:
224:        p1  = buffer1;
225:        p2 = buffer2 + in_size2/2;
226:        p3 = buffer1 + in_size1;
227:
228:
229:        /* The following loop reads in the first half the output image from
230:            the left side image (pointed to by p1) and the second half of the
231:            right side image (pointed to by p2) and combines them in the output
232:            buffer line by line. */
233:
234:        while (p1 < buffer1+in_size1/2)
235:        {
236:
237:            if(p1-buffer1 <= in_size1/2-gap/2)
238:            {
239:                out_line_buf[p1-buffer1]    = *p3;

FILE=a:opaste.c  *  PAGE=4
```

```
240:            }
242:            else
243:            {
244:                out_line_buf[p1-buffer1] = BACKGROUND;
245:            }
246:
247:            if(p1-buffer1 >= gap/2)
248:            {
249:                out_line_buf[(p1-buffer1)+in_size1/2]  = *p2;
250:            }
251:            else
252:            {
253:                out_line_buf[(p1-buffer1)+in_size1/2]  = BACKGROUND;
254:            }
255:
256:            p1++;
257:            p2++;
258:            p3--;
259:        }
260:
261:        fwrite(out_line_buf, sizeof(byte), out_size, out);
262:    }    /*  for each line in the input file  */
263:
264:
265:    fflush(out);
266:
267:    /*  print termination message  */
268:    printf("\n- %s - successfully opasted to - %s - \nand written to - %s -.\n",
269:    input_left, input_right, output);
270:
271:    /*  close files  */
272:    fclose(in1);
273:    fclose(in2);
274:    fclose(out);
275:
276: }  /*  main  */
```

```
 1: /*
 2:              Filename:       slice.c
 3:              Created:        11-SEP-90
 4:              Programmer:     Craig A. Vrana
 5:              Last Modified:  17-SEP-90
 6:              Modified By:    Craig A. Vrana
 7:
 8:          This program recieves the filename of image on the command line.
 9:          It transforms the image so that it has a dual pie-slice window rather
10:          than its normal rectangular window.  A circular area in the center of the
11:          image is set to the value of 128.  The size of this area can be
12:          adjusted by changing the value of DIVISOR at the beginning of the
13:          program.  The size of the area varies inversely to the value of DIVISOR.
14:
15: */
16:
17: #include <stdio.h>
18: #include <string.h>
19: #include <math.h>
20:
21:
22: #define BLACK           0   /* used for the focal point */
23: #define DIVISOR         5   /* divides radius by this to get inner circle radius */
24: #define BACKGROUND  104 /* the shade of grey used to blank portions of the
25:                            image */
26: #define PI M_PI
27:
28:
29: void main(int argc, char *argv[])
30:
31: {
32:     FILE        *file_in,
33:                     *file_out;
34:     float           angle,
35:                     slope;
36:     int         size,
37:                     radius,
38:                     result = NULL,
39:                     x,
40:                     y;
41:     static char     stream_buffer1[16*1024],
42:                     stream_buffer2[16*1024],
43:                     header[64+256],
44:                     in_image_name[64]  = "0",
45:                     out_image_name[64] = "0",
46:                     temp_string[32] = "0",
47:                     val;
48:
49:         /* check if correct number of parameters on command line */
50:     if (argc != 4)
51:     {
52:         printf("Incorrect number of parameters.\n");
53:         printf("Usage: slice input_image output_image angle (.IMG assumed)\n");
54:         return;
55:     }
56:
57:     strcpy(in_image_name, argv[1]);
58:     strcpy(out_image_name, argv[2]);
59:     strcpy(temp_string, argv[3]);

FILE=a:slice.c  *  PAGE=1
```

```
60:
62:     angle = (float)atof(temp_string);
63:     if(angle == 0.0)
64:     {
65:         printf("An angle of 0 degrees would create a blank image\n");
66:         exit();
67:     }
68:
69:     slope = tan((angle/2.0)*PI/180.0);
70:
71:         /*  append .img if necessary  */
72:     if (strrchr(in_image_name, '.')==NULL)
73:     {
74:         strcat(in_image_name, ".img");
75:         strcat(out_image_name, ".img");
76:     }
77:
78:     else if (strrchr(out_image_name, '.')==NULL)
79:     {
80:         strcat(out_image_name, ".img");
81:     }
82:
83:         /* try to open input file */
84:     if ((file_in=fopen(in_image_name, "rb"))==NULL)
85:     {
86:         printf("Unable to open %\n", in_image_name);
87:         return;
88:     }
89:     if (result = setvbuf(file_in, stream_buffer1, _IOFBF,
90:                          sizeof(stream_buffer1)) != NULL)
91:     {
92:         printf("\nIncorrect type or size of buffer1.\n");
93:     }
94:
95:
96:         /* try to open temporary output file */
97:     if ((file_out=fopen(out_image_name, "wb"))==NULL)
98:     {
99:         printf("Unable to open %s\n", out_image_name);
100:        return;
101:    }
102:    if (result = setvbuf(file_out, stream_buffer2, _IOFBF,
103:                         sizeof(stream_buffer2)) != NULL)
104:    {
105:        printf("\nIncorrect type or size of buffer2.\n");
106:    }
107:
108:
109:
110:        /* determine radius of image */
111:    size = 1024 / 2;
112:
113:        /* read in image and write out circular version */
114:    for (y=(-size); y<size; y++)
115:    {
116:        for (x=(-size); x<size; x++)
117:        {
118:            result = fscanf(file_in, "%c", &val);
119:            if(result == EOF)

FILE=a:slice.c  *  PAGE=2
```

```
120:             {
122:                 printf("Got to the end of the image\n");
123:             }
124:             if ((float)x*x+(float)y*y <= (float)size*size &&
125:                 (float)x*x+(float)y*y >= (float)(size/DIVISOR)*(size/DIVISOR))
126:             {
127:                 if(y <= abs((int)(slope*x)) && y >= -abs((int)(slope*x)))
128:                 {
129:                     fprintf(file_out, "%c", val);
130:                 }
131:                 else
132:                 {
133:                     fprintf(file_out, "%c", BACKGROUND);
134:                 }
135:             }
136:             else if ((float)x*x+(float)y*y <= 2)
137:             {
138:                 fprintf(file_out, "%c", BLACK);
139:             }
140:             else
141:             {
142:                 fprintf(file_out, "%c", BACKGROUND);
143:             }
144:         }
145:         if(ferror(file_in) != NULL)
146:         {
147:             printf("\nError reading or writing a file.\n");
148:             break;
149:         }
150:     }
151:     if(ferror(file_in) != NULL)
152:     {
153:         printf("\nError reading or writing a file.\n");
154:     }
155:
156:
157:         /* close files */
158:     fclose(file_in);
159:     fclose(file_out);
160:
161:     printf("Slicing successfully completed.\n");
162:
163: }   /* main */
```

```
/*
            Filename:       vcirc.c
            Created:        02-OCT-89  (From original dated MAY-89)
            Programmer:     Christopher Voltz
            Last Modified:  12-JUN-91
            Modified By:    Craig A. Vrana
        This program recieves the filename of image on the command line.
        It transforms the image so that it has a circular window rather than
        its normal rectangular window.  A circular area in the center of the
        image is set to the value of 128.  The size of this area can be
        adjusted by changing the value of DIVISOR at the beginning of the
        program.  The size of the area varies inversely to the value of DIVISOR.

    Modified on 12-JUN-91:
        Modification made to vastly decrease processing speed from several
        minutes down to approximately 5 seconds.  This was accomplished by
        changing fscanf statements to getc and fprintf statements to putc.
        Also several calculations were moved outside of the most intensive
        loops.

    Modified on 24-OCT-90:
        Modification to make the inner and outer circles on the stimuli
        adjustable.  Changed image size constant to define statement.

    Modified on 27-JUN-90:
        Modification to make the inner circle on the test stimuli to be equal
        to 1/5 the radius of the image. DIVISOR was changed from 4 to 5.
*/

#include <stdio.h>
#include <string.h>
#include <math.h>


#define IMAGE_SIZE  1024        /* Size of one dimension of square image.   */
#define BLACK          0          /* used for the focal point                 */
#define DIVISOR        5        /* divide radius by this to get inner radius  */
#define BACKGROUND  104         /* the level used for the background          */

/* Image section radii */
#define A_INNER     0           /* These values are not correct since the a-  */
#define A_OUTER     100     /* section is always blanked out.               */
#define B_INNER     101
#define B_OUTER     190
#define C_INNER     191
#define C_OUTER     ???
#define D_INNER     ???
#define D_OUTER     380
#define E_INNER     381
#define E_OUTER     511


void main(int argc, char *argv[])

{
    FILE        *file_in,
                    *file_out;
    int         size,
                    radius,

FILE=a:vcirc.c  *  PAGE=1
```

```
                    inner,
                    outer,
                    result = NULL,
                    x,
                    y;
    float           inner_sqrd,
                    outer_sqrd,
                    x_sqrd,
                    y_sqrd;
    static char     stream_buffer1[16*1024],
                    stream_buffer2[16*1024],
                    header[64+256],
                    in_image_name[64]  = "0",
                    out_image_name[64] = "0",
                    inner_string[64] = "0",
                    outer_string[64] = "0",
                    val;

        /* check if correct number of parameters on command line */
    if (argc != 5)
    {
        printf("Incorrect number of parameters.\n");
        printf("Usage: vcirc input_name output_name inner outer\n");
        return;
    }


    strcpy(in_image_name, argv[1]);
    strcpy(out_image_name, argv[2]);
    strcpy(inner_string, argv[3]);
    strcpy(outer_string, argv[4]);

        /*  append .img if necessary  */
    if (strrchr(in_image_name, '.')==NULL)
    {
        strcat(in_image_name, ".img");
        strcat(out_image_name, ".img");
    }

    else if (strrchr(out_image_name, '.')==NULL)
    {
        strcat(out_image_name, ".img");
    }

        /* try to open input file */
    if ((file_in=fopen(in_image_name, "r"))==NULL)
    {
        printf("Unable to open %\n", in_image_name);
        return;
    }
    if (result = setvbuf(file_in, stream_buffer1, _IOFBF,
                         sizeof(stream_buffer1)) != NULL)
    {
        printf("\nIncorrect type or size of buffer1.\n");
    }


        /* try to open output file */
    if ((file_out=fopen(out_image_name, "w"))==NULL)
```

```
120:    {
122:        printf("Unable to open %s\n", out_image_name);
123:        return;
124:    }
125:    if (result = setvbuf(file_out, stream_buffer2, _IOFBF,
126:                         sizeof(stream_buffer2)) != NULL)
127:    {
128:        printf("\nIncorrect type or size of buffer2.\n");
129:    }
130:
131:
132:    inner = atoi(inner_string);
133:    outer = atoi(outer_string);
134:
135:        /* determine radius of image */
136:    size = IMAGE_SIZE / 2;
137:
138:    outer_sqrd = (float)outer*outer;
139:    inner_sqrd = (float)inner*inner;
140:
141:        /* read in image and write out circular version */
142:    for (y=(-size); y<size; y++)
143:    {
144:        y_sqrd = (float)y*y;
145:
146:        for (x=(-size); x<size; x++)
147:        {
148:            x_sqrd = (float)x*x;
149:
150:            val = (char)getc(file_in);
151:            if ((x_sqrd + y_sqrd) <= outer_sqrd &&
152:                 (x_sqrd + y_sqrd) >= inner_sqrd)
153:            {
154:                putc(val, file_out);
155:            }
156:            else if ((x_sqrd + y_sqrd) <= 2)
157:            {
158:                putc(BLACK, file_out);
159:            }
160:            else
161:            {
162:                putc(BACKGROUND, file_out);
163:            }
164:        }
165:        if(ferror(file_in) != NULL)
166:        {
167:            printf("\nError reading or writing a file.\n");
168:            break;
169:        }
170:    }
171:    if(ferror(file_in) != NULL)
172:    {
173:        printf("\nError reading or writing a file.\n");
174:    }
175:
176:
177:        /* close files */
178:    fclose(file_in);
179:    fclose(file_out);

FILE=a:vcirc.c  *  PAGE=3
```

```
180:
182:     printf("vcirc completed successfully.\n");
183:
184: }   /* main */
```

```
 1: /*
 2:                 Program:                scale.c
 3:                 Created:                09-JUL-90
 4:                 Created By:         Craig A. Vrana
 5:                 Last Modified:      10-MAR-92
 6: */
 7:
 8: #include    <get.h>
 9: #include <gl.h>
10: #include <device.h>
11: #include <stdio.h>
12: #include    <math.h>
13:
14:
15: #define C1      4.58
16: #define C2      13.6
17: #define LAM1    0.0061
18: #define LAM2    0.0061
19:
20:
21: char  *infile[32],
22:         *outfile[32];
23: short   image_array[1024][1024];
24:
25: /*-------------------------------------------------------------------------*/
26:
27: main(argc,argv)
28:     int argc;
29:     char *argv[];
30: {
31:
32:     if(argc == 2)
33:     {
34:         strcpy(infile, argv[1]);
35:         if (strrchr(infile, '.')==NULL)
36:         {
37:             strcpy(outfile, infile);
38:             strcat(outfile, "dark");
39:             strcat(outfile, ".img");
40:             strcat(infile, ".img");
41:         }
42:         else
43:         {
44:             printf("Do not enter the .img extension!\n");
45:             exit(0);
46:         }
47:     }
48:     else
49:     {
50:         get_image();
51:         if (strrchr(infile, '.')==NULL)
52:         {
53:             strcpy(outfile, infile);
54:             strcat(outfile, "dark");
55:             strcat(outfile, ".img");
56:             strcat(infile, ".img");
57:         }
58:         else
59:         {

FILE=a:scale.c  *  PAGE=1
```

```
60:             printf("Do not enter the .img extension!\n");
62:             exit(0);
63:         }
64:     }
65:     display_image();
66:     printf("Past display image\n");
67:     scale_image();
68:     printf("Past scale image\n");
69:     save_image();
70:     printf("Past save image\n");
71:     fflush(stdin);
72:
73: }
74:
75:
76: /*-------------------------------------------------------------------------*/
77:
78: display_image()
79: {
80:     int i,j;
81:     FILE *fp;
82:
83:
84:    fp = fopen(infile,"r");
85:     for(i=0; i<1024; i++)
86:     {
87:         for(j=0; j<1024; j++)
88:         {
89:         image_array[i][j] = getc(fp);
90:         }
91:     }
92:     fclose(fp);
93:
94: }
95:
96:
97: /*-------------------------------------------------------------------------*/
98:
99: get_image()
100: {
101:     printf("\nEnter the name of the image to be displayed : ");
102:     scanf("%s", infile);
103:     printf("\n\n");
104: }
105:
106:
107: /*-------------------------------------------------------------------------*/
108:
109: save_image()
110: {
111:
112:     int i,j;
113:     FILE *fp;
114:
115:
116:    fp = fopen(outfile,"w+");
117:     for(i=0; i<1024; i++)
118:     {
119:         for(j=0; j<1024; j++)

FILE=a:scale.c  *  PAGE=2
```

```
120:         {
122:         putc(image_array[i][j], fp);
123:         }
124:     }
125:     fclose(fp);
126: }
127:
128:
129: /*-------------------------------------------------------------------------*/
130:
131: scale_image()
132: {
133:     FILE        *fp;
134:
135:     char        color_string[10];
136:
137:     int     gamma[256],
138:                 high,
139:                 low,
140:                 x,
141:                 y;
142:
143:     double  divisor,
144:                 factor,
145:                 scaled,
146:                 radius;
147:
148:     fp = fopen("/usr1/images/calibration/big_scrn.cal", "r");
149:     for(x=0; x<256; x++)
150:     {
151:         fgets(color_string, 10, fp);
152:         gamma[x] = atoi(color_string);
153:     }
154:     fclose(fp);
155:
156:     divisor = 1.00 / (C1*exp(-LAM1*512.0) + C2*exp(-LAM2*512.0));
157:
158:     for(x=0; x<1024; x++)
159:     {
160:         for(y=0; y<1024; y++)
161:         {
162:             radius = sqrt((double)((x-511)*(x-511)+(y-511)*(y-511)));
163:             if(radius > 512.0)
164:             {
165:                 factor = 1.00;
166:             }
167:             else
168:             {
169:                 factor = 1.00 / (C1*exp(-LAM1*radius) + C2*exp(-LAM2*radius));
170:                 factor /= divisor;
171:             }
172:             scaled = (double)image_array[x][y] * factor;
173:             low = (int)scaled;
174:             high = low + 1;
175:             image_array[x][y] = (short)((scaled-(double)low)*
176:                                         (double)(gamma[high]-gamma[low])+
177:                                         (double)gamma[low]+0.5);
178:         }
179:     }
```

```
180: }
182:
183:
184: /*-----------------------------------------------------------------------*/
```

```
/*                      Program:          segs.c
                        Created by:     Craig A. Vrana - UDRI
                        Date Created:   08-DEC-89
                        Last Modified:  15-JUN-92

        This program was created for Dr. George Geri of the University of
        Dayton Research Institute, Williams Air Force Base, Arizona.

        The purpose of this program is to gather the responses of test
        subjects to segmented image stimuli.

        12-SEP-91
        Modifications were made to the pvrs.c program to get this version.
        The modification made on this date allowed for another subject
        response file to be entered without having to restart the entire
        program.  Also a comment was allowed for that is placed into the
        raw file and also the summary file by the analysis program seg_analyze.

        06-AUG-90
        Modifications were made to the dvs program to get this version.
        The modifications involved the method of data collection and not
        the operating stucture of this program.
*/


/*-------------------------------------------------------------------------*/

                /*****************
                *  Header Files  *
                *****************/

#include <gl.h>
#include <device.h>
#include <stdio.h>
#include    <sys/types.h>
#include    <time.h>


/*-------------------------------------------------------------------------*/

                /*************************
                *  Constant Definitions  *
                *************************/

#define BACKGROUND          157 /* Luminance value to clear screens to. */
#define IMAGE_PRESENTED 20      /* Number of times image appears in test.   */
#define ESC                 27      /* Value of escape character in decimal.    */
#define FIXATION_POINT      "fix92.img\0"   /* Blank image with point.      */
#define LONG                    2       /* For long bell.                       */
#define MAX_NAME_LENGTH 64      /* Length of filenames in characters.       */
#define MAX_TRIALS          1000    /* Arbitrary -- used to size list_array */
#define MIX_FACTOR          3       /* Controls the mix of the random list      */
#define  NUMBER_OF_SCANS    10      /* One scan equals 16.67 ms of display      */
#define REST_DIVISOR        3       /* Controls the number of rest periods. */
#define SHORT                   1       /* For short bell.                      */
#define START_DELAY         3       /* Length of time before test starts.       */


/*-------------------------------------------------------------------------*/
```

```
 60:
 62:                 /*********************
 63:                 *  Global Variables  *
 64:                 *********************/
 65:
 66:           char    list_array[MAX_TRIALS][MAX_NAME_LENGTH],
 67:                   *imagefile[MAX_NAME_LENGTH],
 68:                   *datafile[MAX_NAME_LENGTH],
 69:                   *listfile[MAX_NAME_LENGTH];
 70:           short       image_array[1024][1024];
 71: static  short       *red, *green, *blue;
 72:
 73:
 74: /*-----------------------------------------------------------------------*/
 75:
 76:                 /*********
 77:                 *  main  *
 78:                 *********/
 79:
 80: main(argc,argv)
 81:     int argc;
 82:     char    *argv[];
 83:
 84: {
 85:
 86:     char        correct_response,
 87:                 temp_string[128],
 88:                 comment_string[128];
 89:
 90:     int         i = 0,
 91:                 list_index = 0,
 92:                 trials,
 93:                 trials_between_breaks;
 94:
 95:     long        response;
 96:
 97:     FILE        *output;
 98:
 99:     struct tm   *curtime;
100:
101:     time_t  bintime;
102:
103:
104:     if (argc < 3)       /* If no file names entered on command line.          */
105:     {
106:         ask_file_names();   /* Ask for the list and response file names.      */
107:     }
108:     else
109:     {
110:         strcpy (listfile, argv[1]); /* Get the image listfile name.           */
111:         strcpy (datafile, argv[2]); /* Get the subject datafile name.         */
112:     }
113:
114:     time(&bintime);             /* Get seconds since 00:00:00 GMT, 01-JAN-70  */
115:     curtime = localtime(&bintime);  /* Convert to local time.                 */
116:
117:
118:     trials = get_list();            /* Open listfile, det. # of trials, & load. */
119:     if(trials == NULL)
```

```
120:    {
122:        printf("Aborting due to no stimuli in list file.\n");
123:        exit();
124:    }
125:    randomize_list(trials); /* Create a random list of images to display.   */
126:    trials_between_breaks = (int)(trials/REST_DIVISOR);
127:
128:    add_directory();    /* Prepend '/usr1/images/mean_92/data/' to datafile.    */
129:
130:    while(access(datafile, 0) == NULL)/* Check to see if data file exists.  */
131:    {
132:        printf("Data file already exists.\nChoose another response file.\n");
133:        printf("\nEnter the name of the subject response file : ");
134:        scanf('%s", datafile);
135:        add_directory();    /* Prepend '/usr1/images/mean_92/data/' to datafile*/
136:    }
137:    fflush(stdin);
138:    printf("Enter a comment to be placed in the raw file.\n");
139:    gets(comment_string);
140:
141:    output = fopen(datafile, "w+"); /* Open output datafile.                    */
142:
143:    fprintf(output, "Raw data file name:    %s\n", datafile);
144:    fprintf(output, "Test started:      %s", asctime(curtime));
145:    fprintf(output, "Total trials:      %d\n", trials);
146:    fprintf(output, "Comment:               %s\n\n", comment_string);
147:    fflush(output);
148:
149:    graph_init();       /* Set up the graphics display mode, window, colormap */
150:    clear_screens();    /* Clear the front and back buffs and load fixation.   */
151:
152:    strcpy(imagefile, list_array[list_index++]);    /* Get first image name.    */
153:    strcpy(temp_string, imagefile);                 /* Get name to parse.       */
154:    if(temp_string[2] == '_')
155:    {
156:        correct_response = 'S';
157:    }
158:    else
159:    {
160:        if(temp_string[1] == 's')
161:        {
162:            correct_response = 'R';
163:        }
164:        else
165:        {
166:            correct_response = 'L';
167:        }
168:    }
169:    add_extension();    /* Add ".img" to image name if necessary.                */
170:
171:    load_image();       /* Load the named image into an array in memory.        */
172:
173:    get_start();        /* Wait for the middle mouse button to start test.      */
174:    sleep(START_DELAY);
175:
176:    while(1)                    /* Continue displaying until all images shown.      */
177:    {
178:        if(list_index % trials_between_breaks == 0
179:            && (int)(list_index / trials_between_breaks) != REST_DIVISOR)
```

FILE=a:segs.c * PAGE=3

```
180:        {
182:            get_start();
183:            sleep(3);
184:        }
185:        fputs(imagefile, output);
186:        display_image();    /* Move the image from memory, display and time it.*/
187:        time_display(); /* Display the image for NUMBER_OF_SCANS frames.    */
188:
189:        response = get_response();  /* Get mouse button pushed by subject.      */
190:        switch(response)
191:        {
192:            case LEFTMOUSE:
193:                fputs("\tLeft\t", output);
194:
195:                /* Ring bell only for wrong response when images are different */
196:
197:                if((correct_response != 'L') && (correct_response != 'S'))
198:                {
199:                    ringbell();
200:                    fputs("\tWrong\n", output);
201:                }
202:                else
203:                {
204:                    fputs("\tCorrect\n", output);
205:                }
206:                break;
207:            case RIGHTMOUSE:
208:                fputs("\tRight\t", output);
209:
210:                /* Ring bell only for wrong response when images are different */
211:
212:                if((correct_response != 'R') && (correct_response != 'S'))
213:                {
214:                    ringbell();
215:                    fputs("\tWrong\n", output);
216:                }
217:                else
218:                {
219:                    fputs("\tCorrect\n", output);
220:                }
221:                break;
222:            case ESC:
223:                fclose(output);
224:                graph_reset();
225:                exit();
226:            default:
227:                printf("Value returned from get_response not valid.\n");
228:                graph_reset();
229:                exit();
230:        }
231:
232:        if(list_index < trials) /* If there are more images to display.     */
233:        {
234:            strcpy (imagefile, list_array[list_index++]);
235:            strcpy(temp_string, imagefile);                 /* Get name to parse.*/
236:            if(temp_string[2] == '_')
237:            {
238:                correct_response = 'S';
239:            }
```

```
240:            else
242:            {
243:                if(temp_string[1] == 's')
244:                {
245:                    correct_response = 'R';
246:                }
247:                else
248:                {
249:                    correct_response = 'L';
250:                }
251:            }
252:            add_extension();    /* Add ".img" to image name if necessary.          */
253:
254:            load_image();       /* Load the named image into an array in memory.*/
255:        }
256:        else
257:        {
258:            break;              /* All images have been displayed.                 */
259:        }
260:    }
261:    fclose(output);
262:    graph_reset();          /* Return to original graphics state.                  */
263:
264: }              /*** main ***/
265:
266:
267: /*-------------------------------------------------------------------------*/
268:
269:                  /******************
270:                  *  add_directory  *
271:                  ******************/
272:
273: /*
274:                  USERS:  main
275:
276:                  CALLS:
277:
278:                  PARAM:
279:
280:                  LOCAL:  *temp_string[]
281:
282:                  GLOBL:  *datafile[]
283:
284:                  CONST:  MAX_NAME_LENGTH
285:                              NULL
286: */
287:
288: add_directory()
289: {
290:
291:     char        *temp_string[MAX_NAME_LENGTH];
292:
293:     strcpy(temp_string, datafile);
294:     if(strrchr(temp_string, '.')==NULL)
295:     {
296:         strcat(temp_string, ".raw");
297:     }
298:     strcpy(datafile, "/usr1/images/mean_92/data/");
299:     strcat(datafile, temp_string);
```

```
300:
302: }                   /*** add_directory ***/
303:
304:
305: /*-------------------------------------------------------------------------*/
306:
307:                     /*******************
308:                     *  add_extension  *
309:                     *******************/
310:
311: /*
312:                   USERS:  main
313:
314:                   CALLS:
315:
316:                   PARAM:
317:
318:                   LOCAL:
319:
320:                   GLOBL:  *imagefile[]
321:
322:                   CONST:  NULL
323: */
324:
325: add_extension()
326: {
327:     if (strrchr(imagefile, '.')==NULL)
328:     {
329:         strcat(imagefile, ".img");
330:     }
331:
332: }                   /*** add_extension ***/
333:
334:
335: /*-------------------------------------------------------------------------*/
336:
337:                     /*******************
338:                     *  ask_file_names  *
339:                     ********************/
340:
341: /*
342:                   USERS:  main
343:
344:                   CALLS:
345:
346:                   PARAM:
347:
348:                   LOCAL:
349:
350:                   GLOBL:  *datafile[]
351:                           *listfile[]
352:
353:                   CONST:
354: */
355:
356: ask_file_names()
357: {
358:     printf("\nEnter the name of the image list file : ");
359:     scanf("%s", listfile);
```

```
360:
362:     printf("\nEnter the name of the subject response file : ");
363:     scanf("%s", datafile);
364:
365: }                /*** ask_file_names ***/
366:
367:
368: /*--------------------------------------------------------------------------*/
369:
370:                  /*******************
371:                  *  clear_screens  *
372:                  *******************/
373:
374: /*
375:                  USERS:  main
376:
377:                  CALLS:  load_image
378:                              display_image
379:
380:                  PARAM:
381:
382:                  LOCAL:  *fp
383:
384:                  GLOBL:  *imagefile[]
385:
386:                  CONST:  FIXATION_POINT
387: */
388:
389: clear_screens()
390: {
391:     FILE    *fp;
392:
393:     strcpy(imagefile, FIXATION_POINT);
394:     load_image();              /* Load image from disk into memory.              */
395:     color(image_array[1023][0]);    /* Set the background current color.       */
396:     clear();                   /* Clear the backbuffer with the current color.   */
397:     swapbuffers();
398:     clear();
399:     display_image();           /* Copy image in memory to display memory.        */
400: /*  swapbuffers(); */
401:
402: }                /*** clear_screens ***/
403:
404:
405: /*--------------------------------------------------------------------------*/
406:
407:                  /*******************
408:                  * display_image  *
409:                  *******************/
410:
411: /*
412:                  USERS:  main
413:                              clear_screens
414:
415:                  CALLS:
416:
417:                  PARAM:
418:
419:                  LOCAL:
```

```
420:
422:                GLOBL:  image_array[][]
423:
424:                CONST:
425: */
426:
427: display_image()
428: {
429:
430:     rectwrite(128,0,1151,1023,image_array);
431:     swapbuffers();      /* Swap buffers to display image just written.          */
432:
433:
434: }                /*** display_image ***/
435:
436:
437: /*-------------------------------------------------------------------------*/
438:
439:                 /*************
440:                 *  get_list  *
441:                 ***********/
442:
443: /*
444:                USERS:  main
445:
446:                CALLS:
447:
448:                PARAM:
449:
450:                LOCAL:  i
451:                            index
452:                            trials
453:                            return_value
454:                            *temp_string[]
455:                            *fp
456:
457:                GLOBL:  *listfile[]
458:                            list_array[]
459:
460:                CONST:  MAX_NAME_LENGTH
461:                            IMAGE_PRESENTED
462:                            NULL
463:                            EOF
464: */
465:
466: int get_list()
467: {
468:     int i,
469:             index = 0,
470:             trials = 0,
471:             return_value;
472:
473:     char    *temp_string[MAX_NAME_LENGTH];
474:
475:     FILE    *fp;
476:
477:     if(access(listfile, 4) != NULL)
478:     {
479:         printf("Error: Failed to find %s.\n", listfile);
```

```
480:         abort();
482:     }
483:     fp = fopen(listfile, "r");
484:
485:     do
486:     {
487:         return_value = fscanf(fp, "%s", temp_string);
488:         if(return_value != NULL && return_value != EOF)
489:         {
490:             for(i=0; i<IMAGE_PRESENTED; i++)
491:             {
492:                 trials++;
493:                 strcpy(list_array[index++], temp_string);
494:             }
495:         }
496:     }while(return_value > 0 && return_value != EOF);
497: /*  for (index=0; index<trials; index++)
498:     {
499:         printf("file %d is %s\n", index, list_array[index]);
500:     }*/
501:     fclose(fp);
502:     return(trials);
503:
504: }                   /*** get_list ***/
505:
506:
507: /*--------------------------------------------------------------------------*/
508:
509:                     /******************
510:                     *  get_response  *
511:                     ******************/
512:
513: int get_response()
514: {
515:     short response;
516:     long    value = 0;
517:
518:     qreset();               /* Clear the device queue of extraneous data.       */
519:     while(iqtest());
520:     value = qread(&response);
521: /*  printf("Value = %d  Response = %d\n", value, response);*/
522:     if((value == KEYBD) && (response == ESC))
523:     {
524:         printf("Got escape key\n");
525:         return(ESC);
526:     }
527:
528:     while(value != RIGHTMOUSE && value != LEFTMOUSE)
529:     {
530:         qreset();
531:         while(iqtest());
532:         value = qread(&response);
533:         if((value == KEYBD) && (response == ESC))
534:         {
535:             printf("Got escape key\n");
536:             return(ESC);
537:         }
538:     }
539:     qreset();
```

```
    return((int)value);
}                   /*** get_response ***/


/*-------------------------------------------------------------------------*/

                    /**************
                    *  get_start  *
                    **************/

get_start()
{
    short *response;
    long    value = 0;
    int i;

    setbell(SHORT);
    ringbell();
    for (i=0; i<15; i++)
    {
        gsync();
    }
    ringbell();
    while(!qtest());
    value = qread(&response);
    while(value != MIDDLEMOUSE)
    {
        qreset();
        while(!qtest());
        value = qread(&response);
    }
    setbell(LONG);
    qreset();
}                   /*** get_start ***/


/*-------------------------------------------------------------------------*/

                    /***************
                    *  graph_init  *
                    ***************/

graph_init()
{

    char    color_string[10];

    int     i,
            color_value;

    FILE    *fp;

    prefposition(0,XMAXSCREEN,0,YMAXSCREEN);
    winopen("vres study");
    sleep(1);
    cmode();
    doublebuffer();
    gconfig();
    cursoff();          /* Turn the cursor off.                                 */
```

FILE=a:segs.c * PAGE=10

```
600:
602:     setbell(LONG);          /* Set bell to long beep.                                  */
603:
604:     qdevice(LEFTMOUSE); /* Enable mouse as queued input device.            */
605:     qdevice(MIDDLEMOUSE);
606:     qdevice(RIGHTMOUSE);
607:     qdevice(KEYBD);
608:
609:     save_color_map();
610:
611:     if(access("calibration/vs.cal", 0) != NULL)
612:     {
613:         printf("Cannot find calibration file.\n\n");
614:         printf("Using linear gamma for corrected images.\n\n");
615:         for(i=0; i<256; i++)
616:         {
617:             mapcolor(256+i, i, i, i);
618:         }
619:     }
620:     else
621:     {
622:         fp = fopen("calibration/vs.cal", "r");
623:         ringbell();
624:         printf("Using calibration file for non-corrected images.\n\n");
625:         for (i=0; i<256; i++)
626:         {
627:             fgets(color_string, 10, fp);
628:             color_value = atoi(color_string);
629:             mapcolor(256+i, color_value, color_value, color_value);
630:         }
631:     }
632:
633: }               /*** graph_init ***/
634:
635:
636: /*-------------------------------------------------------------------------*/
637:
638:                 /*****************
639:                 *  graph_reset  *
640:                 *****************/
641:
642: graph_reset()
643: {
644:     restore_color_map();
645:     gexit();
646:
647: }               /*** graph_reset ***/
648:
649:
650: /*-------------------------------------------------------------------------*/
651:
652:                 /****************
653:                 *  load_image  *
654:                 ****************/
655:
656: load_image()
657: {
658:     int i,j;
659:     char    val;
```

```
660:     FILE    *fp;
662:     long    index = 0;
663:
664:     if(access(imagefile, 4) != NULL)
665:     {
666:         printf("Error: Failed to find %s.\n", imagefile);
667:         abort();
668:     }
669:     fp = fopen(imagefile, "r");
670:     for(j=1023; j>(-1); j--)
671:     for(i=0; i<1024; i++)
672:     {
673:         image_array[j][i] = getc(fp)+256;
674:     }
675:     fclose(fp);
676:
677: }                   /*** load_image ***/
678:
679:
680: /*-------------------------------------------------------------------------*/
681:
682:                     /*******************
683:                     *  randomize_list  *
684:                     *******************/
685:
686: randomize_list(trials)
687:     int trials;
688: {
689:     int i,
690:             j,
691:             random_value = 0;
692:     long    seed_value;
693:     char    *temp_string[MAX_NAME_LENGTH];
694:
695:     printf("Total # of trials = %d\n", trials);
696:     seed_value = (long)time(0);
697:     srand(seed_value);
698:     for(j=0; j<MIX_FACTOR; j++)
699:     {
700:         for(i=0; i<trials; i++)
701:         {
702:             random_value =  rand() % (trials);
703:             strcpy(temp_string, list_array[i]);
704:             strcpy(list_array[i], list_array[random_value]);
705:             strcpy(list_array[random_value], temp_string);
706:         }
707:     }
708:
709: }                   /*** randomize_list ***/
710:
711:
712: /*-------------------------------------------------------------------------*/
713:
714:                     /************************
715:                     *  restore_color_map  *
716:                     ************************/
717:
718: restore_color_map()
719: {
```

```
720:     int i;
722:
723:     for(i=0; i<256; i++)
724:     {
725:         mapcolor(i+256, red[i], green[i], blue[i]);
726:     }
727: }
728:
729: /*-------------------------------------------------------------------------*/
730:
731:                     /*******************
732:                     *  save_color_map  *
733:                     *******************/
734:
735: save_color_map()
736: {
737:     int i;
738:
739:     red = (short *)malloc(sizeof(short) * 256 * 3);
740:     green = red + 256;
741:     blue = green + 256;
742:
743:     for(i=0; i<256; i++)
744:     {
745:         getmcolor(i+256, red+i, green+i, blue+i);
746:     }
747: }
748:
749:
750: /*-------------------------------------------------------------------------*/
751:
752:                     /*****************
753:                     *  time_display  *
754:                     *****************/
755:
756: time_display()
757: {
758:     int timer = 0;
759:
760:     while(timer<(NUMBER_OF_SCANS-1))
761:     {
762:         gsync();
763:         timer++;
764:     }
765:     swapbuffers();
766:
767: }                   /*** time_display ***/
768:
769: /*-------------------------------------------------------------------------*/
```

```
/*
                        Created by:     Craig A. Vrana - UDRI
                        Date Created:   12-JAN-90
                        Last Modified:  15-MAY-92

        This program was written to analyze the raw data files produced by
        the variable-resolution study program.  The format of the raw data
        file is as follows:

            Raw data file name:     name.raw
            Test started:               DOW Mon Day Hr:Mn:Sc Year   (DOW = Day of Week)
            Blank line
            Image filename      Response        Result
                                    .
                                    .
                                    .

            Image filename is of the forms:

            zs_s03.img  where 'z' is a letter representing the original image.
                        The 's_s03' denotes that the level 3 image is compared to
                        itself.  The '.img' denotes the file as a binary image
                        file.

            zs03.img    The only difference from the above is that the standard
                        is compared with a portion of the image that has been
                        processed with the variable resolution processing program
                        using an 'a' coefficient of 3.  The name of this image
                        file also denotes that the standard is displayed to the
                        left of the processed image because the 's' is to the left
                        of the processing level number.

            z03s.img    The only difference from the above is that the standard
                        is displayed to the right of the processed portion of the
                        image.

            Additional information can be added to the filename beyond the 4th
            character and before the '.img' suffix without affecting the operation
            of this program up to the point where the filename length reaches the
            limits of the UNIX file system or the constant MAX_NAME_LENGTH.


        Reason modified:

            15-MAY-92
            Added the ability to analyze several data files at one time.

            Changed the operation of the program to calulate the percent
            correct for the new version of the variable resolution study.
            The new vres study program is named vrs.  It places a Left or
            Right response of the subject into the data file instead of
            the original Same or Different.

            Changed the constant MAX_PROCESSING_LEVEL to 100 from 20 to be
            able to handle image numbers up to 99.  Also changed the allign-
            ment on the output to the summary file so that image numbers less
            than 10 allign with the right column of image numbers 10 or greater.
            This was done by placing a field width of 2 in front of the integer
            format character in two fprintf statements.

FILE=a:asegs2.c  *  PAGE=1
```

```
60:
62: */
63:
64: #include <gl.h>
65: #include <device.h>
66: #include <stdlib.h>
67: #include <stdio.h>
68: #include    <ctype.h>
69:
70:
71: /*---------------------------------------------------------------------------*/
72:
73:                  /*************************
74:                  *  Constant Definitions  *
75:                  *************************/
76:
77: #define CORRECT             1
78: #define WRONG                   0
79: #define LEFT                    1
80: #define RIGHT                   0
81: #define MAX_NAME_LENGTH 32      /* Length of filenames in characters.       */
82: #define MAX_TRIALS          10000   /* Arbitrary -- used to size arrays.        */
83: #define READ_ONLY           4       /* Access code for files.                       */
84: #define MAX_PROCESSING_LEVEL        100 /* Levels of processing.                    */
85: #define NUMBER_OF_RESPONSES     2       /* Number of subject responses.     */
86: #define MAX_IMAGES          5       /* Number of possible images in test.       */
87: #define MAX_ANGLES          100
88:
89: /*---------------------------------------------------------------------------*/
90:
91:                  /*********************
92:                  *  Global Variables  *
93:                  *********************/
94:
95: char    name_array[MAX_TRIALS][MAX_NAME_LENGTH],
96:         response_array[MAX_TRIALS][MAX_NAME_LENGTH],
97:         result_array[MAX_TRIALS][MAX_NAME_LENGTH],
98:         *infile[MAX_NAME_LENGTH],
99:         *outfile[MAX_NAME_LENGTH];
100:
101: static int      process_array[MAX_PROCESSING_LEVEL][NUMBER_OF_RESPONSES];
102: static int      angle_array[MAX_ANGLES][NUMBER_OF_RESPONSES];
103: static int      same_array[MAX_PROCESSING_LEVEL][NUMBER_OF_RESPONSES];
104: static float    angles[MAX_ANGLES];
105: static float    temp_angles[MAX_ANGLES];
106:
107: struct image
108: {
109:     char    letter;
110:     int level[MAX_PROCESSING_LEVEL][NUMBER_OF_RESPONSES];
111:     int angle[MAX_ANGLES][NUMBER_OF_RESPONSES];
112:     char    location;
113: };
114: struct image image_array[MAX_IMAGES];
115: struct image standard_array[MAX_IMAGES];
116:
117: /*---------------------------------------------------------------------------*/
118:
119:                  /*********
```

```
120:                    *  main  *
122:                    *********/
123:
124: main(argc,argv)
125:     int argc;
126:     char    *argv[];
127:
128: {
129:     FILE    *input_file,
130:             *output_file;
131:
132:     int angle_length,
133:             return_value,
134:             i,
135:             image = 0,
136:             number_of_angles = 0,
137:             angle_number = 0,
138:             image_number = 0,
139:             index = 0,
140:             files = 1,
141:             int1,
142:             int2,
143:             j,
144:             sum_total,
145:             total,
146:             process_number;
147:
148:     float   temp_angle,
149:             percent_correct,
150:             percent_left;
151:
152:     char    ch1[2],
153:             ch2[2],
154:             angle_string[8],
155:             angle_chars[16],
156:             image_letter[MAX_IMAGES],
157:             name_string[MAX_NAME_LENGTH],
158:             response_string[MAX_NAME_LENGTH],
159:             result_string[MAX_NAME_LENGTH],
160:             temp_string[MAX_NAME_LENGTH],
161:             long_string[128];
162:
163:     if(argc ==1)
164:         ask_file(); /* Ask for the name of the data file to be analyzed.    */
165:
166:     do
167:     {
168:
169:     strcpy (infile, argv[files]);   /* Get name of the data input file.     */
170:     strcpy (outfile, argv[files]);  /* Get name of the data output file.    */
171:
172:     add_extensions();
173:
174:     if(access(infile, READ_ONLY) != NULL)
175:     {
176:         printf("Error: Failed to find %s\n", infile);
177:         exit();
178:     }
179:     input_file = fopen(infile, "r");
```

FILE=a:asegs2.c * PAGE=3

```
180:     output_file = fopen(outfile, "w+");
182:
183:
184:         /* Copy first five lines from .raw file to .sum file. */
185:     fgets(long_string, 100, input_file);
186:     fputs(long_string, output_file);
187:     fgets(long_string, 100, input_file);
188:     fputs(long_string, output_file);
189:     fgets(long_string, 100, input_file);
190:     fputs(long_string, output_file);
191:     fgets(long_string, 100, input_file);
192:     fputs(long_string, output_file);
193:     fgets(long_string, 100, input_file);
194:     fput (long_string, output_file);
195:     fflush(output_file);    /* Flush stream to stop doubling of lines. */
196:
197:
198:         /* Initialize the char string that contains everything in an angle. */
199:     strcpy(angle_chars, "0123456789,\0");
200:
201:
202:
203:         /* Load arrays from .raw file and close .raw file. */
204:     do
205:     {
206:         return_value=fscanf(input_file, "%s %s %s", name_string, response_string,
207:                                                                 result_string);
208:
209:         if(return_value != NULL && return_value != EOF)
210:         {
211:             strcpy(name_array[index], name_string);
212:             strcpy(response_array[index], response_string);
213:             strcpy(result_array[index], result_string);
214:             index++;
215:         }
216:
217:         /* Get all the different image letters and count how many there are.    */
218:         for(i=0; i<MAX_IMAGES; i++)
219:         {
220:             if((name_string[0] != image_array[i].letter) && (i < image_number))
221:             {
222:                 continue;
223:             }
224:             else if(name_string[0] == image_array[i].letter)
225:             {
226:                 break;
227:             }
228:             else
229:             {
230:                 standard_array[image_number].letter = name_string[0];
231:                 image_array[image_number++].letter = name_string[0];
232:                 break;
233:             }
234:         }
235:
236:         /* Get all the different image segments and count how many there are.*/
237:         for(i=0; i<MAX_ANGLES; i++)
238:         {
239:
```

```
240:            if(name_string[4] == '_')
242:            {
243:                angle_length = strspn(&name_string[5], angle_chars);
244:                for(j=0; j<angle_length; j++)
245:                {
246:                    angle_string[j]=name_string[5+j];
247:                    if (angle_string[j] == ',')
248:                        angle_string[j] = '.';
249:                }
250:                angle_string[j] = '\0';
251:                temp_angle = (float)atof(angle_string);
252:
253:            }
254:
255:            else if(name_string[6] == '_')
256:            {
257:                angle_length = strspn(&name_string[7], angle_chars);
258:                for(j=0; j<angle_length; j++)
259:                {
260:                    angle_string[j]=name_string[7+j];
261:                    if (angle_string[j] == ',')
262:                        angle_string[j] = '.';
263:                }
264:                angle_string[j] = '\0';
265:                temp_angle = (float)atof(angle_string);
266:
267:            }
268:
269:            else
270:            {
271:                printf("Unknown data found when scanning for segments\n");
272:            }
273:
274:
275:            if((temp_angle != temp_angles[i]) && (i < number_of_angles))
276:            {
277:                continue;
278:            }
279:            else if(temp_angle == temp_angles[i])
280:            {
281:                break;
282:            }
283:            else
284:            {
285:                temp_angles[number_of_angles++] = temp_angle;
286:                break;
287:            }
288:
289:        }
290:
291:
292:    }while(return_value > 0 && return_value != EOF);
293:
294:
295:    fclose(infile);
296:
297:    }while(files++ <argc);
298:
299:
```

```
300:     /* Sort the angles into ascending order */
302:
303:     for(i=0; i<number_of_angles-1; i++)
304:     {
305:         for(j=i+1; j<number_of_angles; j++)
306:         {
307:             if(temp_angles[j] < temp_angles[i])
308:             {
309:                 temp_angle = temp_angles[j];
310:                 temp_angles[j] = temp_angles[i];
311:                 temp_angles[i] = temp_angle;
312:             }
313:         }
314:         angles[i] = temp_angles[i];
315:     }
316:
317:     angles[i] = temp_angles[i];
318:
319:
320:
321:     /* Tally the results of all images from the loaded arrays. */
322:
323:
324:     for(i=0; i<index; i++)      /* For every trial  */
325:     {
326:         strcpy(temp_string, name_array[i]);
327:         if(temp_string[4] == '_')
328:         {
329:             angle_length = strspn(&temp_string[5], angle_chars);
330:             for(j=0; j<angle_length; j++)
331:             {
332:                 angle_string[j]=temp_string[5+j];
333:                 if (angle_string[j] == '_')
334:                     angle_string[j] = '.';
335:             }
336:             angle_string[j] = '\0';
337:             temp_angle = (float)atof(angle_string);
338:             /*  printf("angle %d = %5.1f\n", i, temp_angle);  */
339:         }
340:         else if(temp_string[6] == '_')
341:         {
342:             continue;
343:         }
344:         else
345:         {
346:             printf("Unknown data found when scanning input\n");
347:             exit(-1);
348:         }
349:
350:
351:
352:         for(j=0; j<number_of_angles; j++)
353:         {
354:             if(angles[j] == temp_angle)
355:             {
356:                 angle_number = j;
357:             }
358:         }
359:
```

```
360:
362:
363:        if(strcmp(result_array[i], "Correct") == NULL)
364:        {
365:            ++angle_array[angle_number][CORRECT];
366:            for(j=0; j<image_number; j++)
367:            {
368:                if(image_array[j].letter == temp_string[0])
369:                {
370:                    ++image_array[j].angle[angle_number][CORRECT];
371:                    break;
372:                }
373:            }
374:        }
375:        else
376:        {
377:            ++angle_array[angle_number][WRONG];
378:            for(j=0; j<image_number; j++)
379:            {
380:                if(image_array[j].letter == temp_string[0])
381:                {
382:                    ++image_array[j].angle[angle_number][WRONG];
383:                    break;
384:                }
385:            }
386:        }
387:    }
388:
389:
390:        /* Put headers in the output file.              */
391:    fprintf(output_file, "\t\t\t          Total");
392:    for (i=0; i<image_number; i++)
393:    {
394:        fprintf(output_file, "\t   %c", image_array[i].letter);
395:    }
396:    fprintf(output_file, "\n");
397:
398:
399:        /* Calculate percentages from tallies and store in .sum file. */
400:
401:    for(i=0; i<number_of_angles; i++)
402:    {
403:        if(angle_array[i][CORRECT]==0 && angle_array[i][WRONG]==0)
404:        {
405:            continue;
406:        }
407:        total = angle_array[i][CORRECT] + angle_array[i][WRONG];
408:        percent_correct = (float)angle_array[i][CORRECT] / (float)total;
409:        percent_correct *= 100.00;
410:
411:        fprintf(output_file, "Seg >: %5.1f  Percent Correct: %6.2f",
412:                angles[i], percent_correct);
413:
414:        sum_total = 0;
415:        for(j=0; j<image_number; j++)
416:        {
417:        total = image_array[j].angle[i][CORRECT] +
418:                image_array[j].angle[i][WRONG];
419:        percent_correct = (float)image_array[j].angle[i][CORRECT] / (float)total;
```

```
420:         percent_correct *= 100.00;
422:         sum_total += total;
423:         fprintf(output_file, "\t%6.2f",percent_correct);
424:         }
425:         fprintf(output_file, "\t%d trials\n", sum_total);
426:     }
427:
428:
429:     /* Tally the results of the same images from the loaded arrays. */
430:
431:
432:     for(i=0; i<index; i++)
433:     {
434:         strcpy(temp_string, name_array[i]);
435:         if(temp_string[6] == '_')
436:         {
437:             angle_length = strspn(&temp_string[7], angle_chars);
438:             for(j=0; j<angle_length; j++)
439:             {
440:                 angle_string[j]=temp_string[7+j];
441:                 if (angle_string[j] == ',')
442:                     angle_string[j] = '.';
443:             }
444:             angle_string[j] = '\0';
445:             temp_angle = (float)atof(angle_string);
446:
447:         }
448:         else
449:         {
450:             continue;
451:         }
452:
453:
454:         for(j=0; j<number_of_angles; j++)
455:         {
456:             if(angles[j] == temp_angle)
457:             {
458:                 angle_number = j;
459:             }
460:         }
461:
462:         if(strcmp(response_array[i], "Left") == NULL)
463:         {
464:             ++same_array[angle_number][LEFT];
465:             for(j=0; j<image_number; j++)
466:             {
467:                 if(standard_array[j].letter == temp_string[0])
468:                 {
469:                     ++standard_array[j].angle[angle_number][LEFT];
470:                     break;
471:                 }
472:             }
473:         }
474:         else
475:         {
476:             ++same_array[angle_number][RIGHT];
477:             for(j=0; j<image_number; j++)
478:             {
479:                 if(standard_array[j].letter == temp_string[0])
```

```
480:                {
482:                    ++standard_array[j].angle[angle_number][RIGHT];
483:                    break;
484:                }
485:            }
486:        }
487:    }
488:
489:    fprintf(output_file, "\n\nSame image comparison\n");
490:
491:        /* Calculate percentages from tallies and store in .sum file. */
492:
493:    for(i=0; i<number_of_angles; i++)
494:    {
495:        if(same_array[i][LEFT]==0 && same_array[i][RIGHT]==0)
496:        {
497:            continue;
498:        }
499:        total = same_array[i][LEFT] + same_array[i][RIGHT];
500:        percent_left = (float)same_array[i][LEFT] / (float)total;
501:        percent_left *= 100.00;
502:
503:
504:        fprintf(output_file, "Seg >: %5.1f  Percent Left:    %6.2f",
505:                    angles[i], percent_left);
506:
507:
508:        sum_total = 0;
509:        for(j=0; j<image_number; j++)
510:        {
511:        total = standard_array[j].angle[i][LEFT] +
512:                    standard_array[j].angle[i][RIGHT];
513:        percent_left = (float)standard_array[j].angle[i][LEFT] /
514:                                                                    (float)total;
515:        percent_left *= 100.00;
516:        sum_total += total;
517:        fprintf(output_file, "\t%6.2f",percent_left);
518:        }
519:        fprintf(output_file, "\t%d trials\n", sum_total);
520:    }
521:
522:    fclose(outfile);
523:
524:
525: }               /*** main ***/
526:
527: /*-------------------------------------------------------------------------*/
528:
529:                 /*************
530:                 *  ask_file  *
531:                 ***********/
532:
533: ask_file()
534: {
535:     printf("\nEnter the name of the data file to be analyzed (.raw) : ");
536:     scanf("%s", infile);                    /* Get name of the raw data file.      */
537:     strcpy (outfile, infile);           /* Get name of the data output file.    */
538: }               /*** ask_file ***/
539:
```

```
540:
542: /*------------------------------------------------------------------------*/
543:
544:                     /*******************
545:                     *  add_extensions  *
546:                     *******************/
547:
548: add_extensions()
549: {
550:     if (strchr(infile, '.')==NULL)
551:     {
552:         strcat(infile, ".raw");
553:         strcat(outfile, ".sum");
554:     }
555:     else
556:     {
557:         /* strip extension from input file and add extension to output */
558:         strncpy(outfile, infile, strlen(infile)-4);
559:         strcat(outfile, ".sum");
560:     }
561:
562: }                   /*** add_extensions ***/
563:
564:
565: /*------------------------------------------------------------------------*/
```

APPENDIX C

The program, *mtf.c*, used to calculate the MTFs of the wide-field display used in the present study. The program was written to implement the procedures for calculating the MTF described by Kelly (1992).

```
 1:
 2:
 3:
 4: /*************************************************************************
 5:
 6:                 MTF.C
 7:
 8: *************************************************************************/
 9:
10: #include <stdio.h>
11: #include <stdlib.h>
12: #include <math.h>
13: #include <graph.h>
14: #include <string.h>
15: #include <conio.h>
16: #include "dft3.c"
17: #define PI 3.1415926535898
18:
19: void dft1(int,int);
20: void graph_setup(void);
21: void axes_positive(void);
22: void plot_discxy(float [], float [], float []);
23: void plot_contxy();
24: void close_graph(void);
25: struct videoconfig vc;
26:
27: double dft_r[1024], dft_i[1024];
28:
29: main(argc,argv)
30: int argc;
31: char *argv[];
32: {
33:     int             i,
34:                     imax,
35:                     j,
36:                     temp,
37:                     tempi,
38:                     length,
39:                     display_lines,
40:                     separation,
41:                     npts,
42:                     StrLength,
43:                     arg;
44:
45:     static float    x[512],
46:                     data[512],
47:                     input = (float)0.0,
48:                     scale[4],
49:                     tempr,
50:                     sqwrx[10],
51:                     sqwry[10];
52:
53:     static float    contrast_ratio,
54:                     dc_modulation,
55:                     y_gain,wing_min = (float)0.0;
56:
57:     static double   calib,
58:                     calratio,
59:                     half_calib,

FILE=a:kellymtf.c  *  PAGE=1
```

```
60:                     pixels_height=1000.0,
62:                     sinc_arg,
63:                     pad,
64:                     slope,
65:                     tempd,
66:                     meas_data[128],
67:                     filt_data[500],
68:                     freq_step;
69:
70:
71:     FILE            *fptr;
72:
73:     static char     chtemp[16],
74:                     string[28];
75:
76:     static double   centroid1 = (double)0.0,
77:                     centroid2 = (double)0.0,
78:                     sum1 = (double)0.0,
79:                     sum2 = (double)0.0,
80:                     peak1,
81:                     peak2;
82:
83:     double          dummy2 = 1.23456789,
84:                     dummy3 = 2.345678,
85:                     dummy1;
86:
87:     static char     name1[25],
88:                     name2[25],
89:                     name3[25];
90:
91:
92: /*************************************************************************
93:     FREQUENCY SCALING CALIBRATION
94: *************************************************************************/
95:     if(argc != 6)
96:     {
97:         printf("Usage: mtfnew raw_file cal_file scrn_height line_sep c_ratio\n");
98:         return(-1);
99:     }
100:
101: /*
102:     else
103:     {
104:         get_info();
105:     }
106: */
107:
108:     strcpy(name1,argv[1]);
109:     strcpy(name2,argv[2]);
110:     display_lines = atoi(argv[3]);
111:     separation = atoi(argv[4]);
112:
113:
114:     /* 1.  Read in 111 calibration points. */
115:
116:     fptr = fopen(name2,"r");
117:
118:     for (i=0; i<111; i++)
119:     {
```

```
120:          fscanf(fptr, "%*f %f", &input);
122:          meas_data[i] = (double)input;
123:          printf("%8.4lf\n",meas_data[i]);
124:      }
125:      fclose(fptr);
126:      printf("Closed the calibration file.\n");
127:
128:      /* 2.  Find the pixel or element with the largest value in the
129:         first 56 elements. */
130:
131:      for (i=0; i<56; i++)
132:      {
133:          if(meas_data[i] > peak1)
134:          {
135:              peak1 = meas_data[i];
136:          }
137:      }
138:      peak1 /= 2;
139:
140:      /* 3.  Set all elements in the first 56 elements to zero,
141:          which are less than half of the peak value found in step 2. */
142:
143:      for (i=0; i<56; i++)
144:      {
145:          if (meas_data[i] < peak1)
146:          {
147:              meas_data[i] = (double)0;
148:          }
149:
150:          /* 4.  Sum the value of each element multiplied by its element
151:              number(i.e. 1-56). */
152:
153:          centroid1 += meas_data[i]*(double)(i);
154:
155:          /* 5.  Sum the value of each of the first 56 elements. */
156:
157:          sum1 += meas_data[i];
158:      }
159:
160:      /* 6.  Divide the sum calculated in step 4 by the sum calculated in
161:          step 5.  This number is the centroid of the peak in fractions of
162:          elements. */
163:
164:      centroid1 /= sum1;
165:
166:
167:      /* 7.  Repeat steps 2 through 6 for the last 55 elements. */
168:
169:      for (i=56; i<111; i++)
170:      {
171:          if(meas_data[i] > peak2)
172:          {
173:              peak2 = meas_data[i];
174:          }
175:      }
176:      peak2 /= 2;
177:
178:
179:      for (i=56; i<111; i++)
```

```
180:     {
182:         if (meas_data[i] < peak2)
183:         {
184:             meas_data[i] = (double)0;
185:         }
186:         centroid2 += meas_data[i]*(double)(i-56);
187:         sum2 += meas_data[i];
188:     }
189:     centroid2 = centroid2/sum2 + (double)56.00;
190:
191:
192:
193:     calratio = (double)display_lines / (double)separation;
194:
195:     /* 8.  Take the diffence between the two centroids.  This is the
196:        distance between the two calibration peaks in units of sensor pixels
197:        or elements. */
198:
199:     /* 9.  Perform the following calculation:
200:
201:                                centroid diff. *  lines/picture height
202:     Calibration constant = ------------------- -----------------
203:                                display lines between cal. lines * 512
204:     */
205:
206:     calib = ((centroid2 - centroid1) * calratio) / (double)512.0;
207:
208:     npts = (int)(((double)display_lines+256.000)/calib+0.5);
209:
210:     printf("\n\n%8.4e\n%8.4e\n",centroid1,centroid2);
211:
212: /*************************************************************************
213:     INPUT DATA
214: *************************************************************************/
215:
216:     fptr = fopen(name1,"r");
217:
218:     for (i=0; i<111; i++)
219:     {
220:         if(fscanf(fptr, "%*f %f", &input) == 0)
221:         {
222:             printf("Ran out of input data before finished reading file.\n");
223:             exit(-1);
224:         }
225:         data[i] = input;
226:         printf("%8.4lf\n",data[i]);
227:     }
228:
229:     fclose(fptr);
230:
231: /*************************************************************************
232:     APPLY TRAPEZOIDAL WINDOW FUNCTION
233: *************************************************************************/
234:
235:
236:     for(i=0; i<512; i++)
237:     {
238:         dft_i[i] = (double)0;
239:         dft_r[i] = (double)0;
```

```
240:     }
242:
243:
244: /*************************************************************************
245:     PAD DATA and GRAPH
246: *************************************************************************/
247:
248:     wing_min = min(data[0],  data[110]);
249:
250:     for (i=0; i<111; i++)
251:     {
252:         data[i] = data[i] - wing_min;
253:     }
254:
255:     for (i=0; i<40; i++)
256:     {
257:         data[i] = data[i]*((float)(i))*0.025;
258:         data[110-i] = data[110-i]*((float)(i))*0.025;
259:     }
260:
261:     for (i=200; i<311; i++)
262:     {
263:         dft_r[i] = (double)(data[i-200]);
264:     }
265:
266:     for (i=0; i<200; i++)
267:     {
268:         dft_r[i] = (double)(0);
269:     }
270:
271:     for (i=311; i<512; i++)
272:     {
273:         dft_r[i] = (double)(0);
274:     }
275:
276:
277: /*************************************************************************
278:     CALL DISCRETE FOURIER TRANSFORM ROUTINE
279: *************************************************************************/
280:
281:     dft1(1,npts);
282:
283:
284: /*************************************************************************
285:     SCALE TRANSFORM & CALCULATE AMPLITUDE
286: *************************************************************************/
287:
288:     /* Big number for 0 dark; Light / Dark*/
289:
290:     contrast_ratio = (float)atof(argv[5]);
291:     printf("Got to contrast ratio calculation.\n");
292:
293:     dc_modulation = (contrast_ratio - 1.0) / (contrast_ratio + 1.0);
294:
295:     for (i=0; i<npts; i++)
296:     {
297:         x[i] = (float)calib*(float)(i);
298:         data[i] = (float)(sqrt(dft_r[i]*dft_r[i] + dft_i[i]*dft_i[i]));
299:         if(i==0)
```

```
300:         {
302:             y_gain = dc_modulation/data[0];
303:             data[0] = dc_modulation;
304:         }
305:         else
306:             {
307:             data[i] = data[i]*y_gain;
308:             }
309:
310:     }
311:
312:
313:     freq_step = atof(argv[3]);
314:     for (i=0; i<npts; i++)
315:         {
316:         if (x[i] > (float)freq_step)
317:             {
318:             imax = i + 1;
319:             i = npts;
320:             }
321:         }
322:
323:
324:     StrLength = strcspn(argv[1],".");
325:     printf("StrLength = %d\n", StrLength);
326:     strncpy(name3,argv[1],StrLength); /* copy name only */
327:     printf("FileName = %s\n", name3);
328:
329:
330:
331:     strcat(name3,".dat");
332:     if((fptr = fopen(name3,"wt")) == NULL)
333:     {
334:         printf("Error opening %s for writing... program aborting.\n", name3);
335:         exit(-1);
336:     }
337:     fputs(argv[1],fptr);
338:     fputs("\n",fptr);
339:     fprintf(fptr,"%d\n",imax);
340:     for (i=0; i<imax; i++)
341:     {
342:         fprintf(fptr,"%20.3f %20.3f\n",x[i],data[i]);
343:     }
344:     fclose(fptr);
345:
346:
347: }
```