A Real-Time Three Dimensional Profiling Depth from Focus Method

by

James Wiley Fleming

Submitted to the Department of Electrical Engineering and Computer Science in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 1994

Author .... [illegible]
Department of Electrical Engineering and Computer Science
May 18, 1994

Certified by .... [illegible]
Gill Pratt
Assistant Professor
Thesis Supervisor

Accepted by .... [illegible]
Frederic R. Morgenthaler
Chairman, Departmental Committee on Graduate Students

A Real-Time Three Dimensional Profiling Depth from Focus Method

by

James Wiley Fleming

Submitted to the Department of Electrical Engineering and Computer Science
on May 18, 1994, in partial fulfillment of the
requirements for the degree of
Master of Science in Computer Science and Engineering

Abstract

The surface profile of a three dimensional scene can be determined using a depth from focus method. The scene is imaged at multiple focal depths with a shallow depth of focus. Range information is extracted from the images by finding the focal depth of maximal contrast, determined by fitting a curve to a measure of contrast. From the focal depth and imaging system optical parameters, the actual distance can be derived. As well, a single all in focus image can be constructed of the scene.

On the assumption that most good contrast data comes from edges, a mathematical model for a defocused and pixel quantized half plane step response is developed, along with a model of its theoretical contrast curve.

The methods explored here are for a passive system, that is, one that does not change the scene by projecting a dot, grid, or line pattern onto the scene with a projector in order to impose contrast on the scene. Passive systems are criticized for their failure in scenes with low contrast. Real time error correction methods are examined that allow data in these low contrast areas to be reconstructed from surrounding depth data in high contrast areas. This allows us to use a small 3x3 contrast summation window, which is more responsive to quickly changing depth slopes than the much larger contrast summation windows that are typically used.

The algorithms are applied to experimentally gathered images of objects on a table, a scene with relatively low contrast information except around edges.

The algorithms developed are done so with an eye toward a real time hardware implementation, allowing full depth analysis of scenes at multiple frames per second. Such a system could be useful in reconstructing a stereo image in real time situations where you can only have a single lens because of space constraints, such as in laproscopic surgery.

Thesis Supervisor: Gill Pratt
Title: Assistant Professor

Acknowledgments

First I would like to thank Professor Pratt for his help in this thesis, and for prodding me to tackle the nasty problems that I would have rather ignored.

Next I would like to thank my dog Oscar for providing me with a sound philosophical basis on which to base my life.

Thanks to the Coca Cola company for their fine line of thesis writing products.

Thanks to Benson Wen for camera consulting.

Thank you to my pal Jon Maiara for years of companionship, craziness, and the use of his micrometer.

Thank you to my new family, for bringing me hope.

Finally, a very special thank you to my parents, for emotional support, encouragement, and always believing in me.

In memory of my mother,

Dr. Marianne Fleming.

Contents

List of Figures

Chapter 1

Introduction

Previous depth from focus methods for acquiring surface profiles have concentrated on imaging homogenous materials lit with a projected pattern of dots or a grid. These methods have been computationally intensive. We can eliminate the need for a projected light pattern and apply depth from focus techniques in real time using parametric blur curve fitting and data dependent error correction techniques. We will also explore methods of extracting a single all in focus picture of the scene.

These techniques explored here are designed to be used when you only have one camera, with a large aperature and hence a shallow depth of focus. For instance in laproscopic surgery where there is only room for one lens. If you have space for two cameras, you could use one with a small aperature to provide the all in focus picture, and use the other for depth information, or you could use both with narrow aperatures and use stereoscopic depth finding.

To generate more contrast in a scene you could use one lens to project a patterned light source onto the scene. This is called an active system, active because it changes the scene it is measuring. The disadvantage of such systems is that they *have* to change the scene by imposing their own lighting. In controlled environments, such as on an assembly line, these methods are very effective. Outdoors, in an area already brightly lit, or in any other situation where you need homogenous lighting, you can't have the luxury of using your own special patterned light source.

As well, they share the disadvantage that the depth map resolution is limited to the resolution of the projected pattern, and not to the resolution of the scanned images. With a passive system, your depth map resolution is limited only to the resolution of the orginal scanned images. We will be researching ways to get close to the limit.

The passive method of depth from focus has been criticized as ineffective because it depends on natural scene detail to work. For instance, a white blank wall is going to look like a white blank wall whether it is in focus or not. However, scenes of any interest are likely to have detail in them somewhere. As well, there are often flaws or detail somewhere on an otherwise smooth surface that can tell us the depth of that surface.

Given a single camera system, there are three ways such a system is focused. The most common, which we find in standard 35mm cameras and most video cameras is the movable lens system. In such a system you hold the film steady and move the lens in order to focus. This is perhaps the least suitable way to focus, as changing the distance from lens to object changes the perspective of the scene. Objects near the camera will appear larger in proportion to distant objects when the camera is focused near than when it is focused far.

A better way is to keep the lens fixed in space and vary the distance of the film to the lens. We will call this a fixed lens system. in such a system perspective is constant, so that all parts of an image are in the same relative scale to each other no matter where the scale is. However, you still have the problem that at different focal distances your images are at different scales. The farther the film is from the lens, the larger the image will be. So that at when focused near your images are bigger than when focused far away. This scale change needs to be compensated for.

Perhaps the best kind of system is the variable focal length lens system, one similar to our eyes. In our eyes, both the lens and retina remain fixed, and to focus we change the focal length of the lens by changing the shape of the lens. The Alvarez lens is such a lens, currently used in Polaroid cameras. The chief advantage of such a lens is that images taken with such a lens are all at the same scale, no matter where the focus is. In the other two systems the scale of the incoming images changes depending on whether its focused near or far. We'll be exploring the mathematics of these systems later.

A word about optimization. These days the techniques of optimization are different from those in the past. Floating point units are as fast or faster than the corresponding integer math processors; so that multiplies, adds, and indeed memory access all occur at the same speed. We group these three operations together and call them all "operations." Our goal in optimizing is to reduce the total number of these operations and to try avoiding the use of division and functions approximated by series, like roots and exponentials, unless they can be table driven.

Also, with more and more parallel machines appearing, we should be keeping an eye out for readily parallelizable algorithms.

I wrote all the original code for these algorithms using Matlab, an interpreted math language designed for matrix manipulation. The code does not run super fast, but is very high level, has vectors and matrices as basic data types and was easily coerced into doing image processing. Code is easily changed and rerun, and many fundamental signal processing operators already exist. The code is very easy to follow, easy to port, and explicitly demonstrates the algorithms as much as pseudo-code would. I've included some of the more relevant Matlab functions that I wrote in Appendix B.

Chapter 2

Background

The earliest work with depth from focus techniques, see [5, 6, 7, 9], was done as a non-scratching substitute for depth measuring diamond styli used in industrial quality control. These systems were essentially optical styli. A lens mechanism slewed across the object, focused by an analog feedback system, its position recorded by a pen. Some of these systems were passive, focusing on the specular glint of the polished metal, while others were active, projecting lines onto the surface. The best success was had with the active systems, as diffuse objects might have little to focus on, and small defects on the object being scanned could cause glare to screw up the system.

Engelhardt and crew [8] improved on the efficiency of this scanning system by using a television camera to do the imaging. Their system was used to sense the position of an item on an assembly line so that a robot arm could manipulate it. Except for a memory, the processing in this system was analog. This system was also active, projecting lines onto the object being scanned.

Stuart [1], Delisle [2], and Fieguth [4] built further upon the optical stylus idea, implementing digital processing image systems rather than analog ones, again using a projected pattern for best results. These systems were all aimed at industrial quality control. Delisle and Fieguth both fit blur curve templates to the data. Stuart best-fit a second order polynomial to the tops of his blur curves to find the peak but required tens or hundreds of measurements at different focal distances. Both methods are slow.

An interesting and apparently quite effective twist on these active methods was developed by Stephen Scherock [3], who used a pinhole camera for imaging (keeping the whole scene totally in focus) and then projects *the pattern* with a limited depth of field. By computing the amount of blur compared to the original pattern, he got an entire depth map from just one sample. This method was developed for use in machine vision, more for manuevering around rooms and manipulating objects than for industrial quality control.

Shree Nayar [16] developed a working system for doing industrial control type surface profiling of uniformly textured objects. He has a nice treatment of various contrast measuring metrics, and settles on the Modified Sum Laplacian as the measure of choice for uniformly textured objects.

Reg Willson and Steve Shafer [10, 11] out at CMU have done some nice work on camera centering characterization and color register correction. The RGB planes of an image are imaged at different focal points and scale because the index of refraction of a typical lens varies so much with wavelength. Buyer beware.

Yalin Xiong and Steven Shafer [12] developed a very accurate method for getting depth from defocus, using only two images, calculating how to blur one to get the other. Knowing how much to blur it corresponds to knowing how far the object image is from the plane of best focus, which can be mapped into how far the object is from the lens. This is very accurate, but requires iterative solutions at every depth point. One would gather not a real time process.

Subbarao with T. Choi [15] get depth information in the usual way, and then best fit planar segments to the data to get a good shape model. Easily extensible to fit parabolic or more complex shape models.

Subbarao and Ming-Chin Lu [14] developed a software system for simulating and modeling optical defocus and acquisition.

There was actually a lot more work done on this problem than I originally anticipated, and the thrust of this thesis has been correspondingly shifted towards data dependent error correction, detailed contrast curve analysis, and keeping the convolution window small.

Chapter 3

Review of Geometrical Optics

We are going to review basic geometrical optics, and what they imply for scale and focus.

A basic imaging system has a lens and an imaging device like film or a ccd array. A lens has a focal length f, which determines how far from the lens an object so distance away appears in focus behind the lens. The focal length is the number you usually hear when describing a lens. For instance, a 50mm lens for your camera has a focal length of 50mm. The relationship between focal length, object distance and image distance is:

$$\frac{1}{so} + \frac{1}{si} = \frac{1}{f} \tag{3.1}$$

Note that si is positive behind the lens and so is positive in front of it, so to focus on an object so in front of the lens, you want to put the film

$$si = \frac{1}{\frac{1}{f} - \frac{1}{so}}$$

behind the lens.

From this you see that closest the object can be to the lens is $so = f$, because at that point the image is in focus infinity away from the lens. Conversely, when the object is infinitely far away, you focus on it by placing the film the focal distance away from the lens: $si = f$.

When generating a depth map we are going to want to know the distance d of any part of the image relative to a fixed point. In a fixed film setup that distance will be from the film to the object, which will be:

$$d = si + so$$

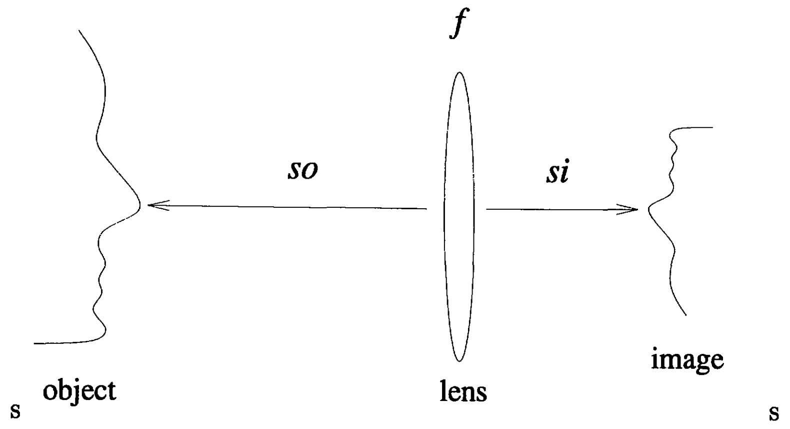


Figure 3-1: Lens with object and projected image.

In a fixed lens system and the variable focal length lens system, the lens remains fixed in space and we measure distance relative to that:

$$d = so$$

3.1 Changing Scale

The scale at which an image appears on film depends upon the size of the object, the distance of the object from the lens, and the distance of the film from the lens. We can trace two rays from the object, one going flat through the center, and one slanting down from the top of the object, through the center of the lens and onto the image plane. The thin lens model says that all rays traced through the middle of the lens pass through unbent, which is why we can do this.

We introduce a new variable now, sl, which is the distance the lens is from the imaging device. When sl equals si, your image is in focus.

The height of the object is ho, the height of the image is hi. The magnification factor of the image is the ratio of the image height to the object height, or hi/ho. Usually this will be less than one[1].

[1]But not always! The C terminal at Logan Airport in Boston has a nice display of six gigantic polaroids of an ancient chinese silk scroll painting. They used a room sized camera, and the polaroids are each 8 feet tall and 3 feet wide, and have a magnification of about ten. Its very excellent, you can clearly see every strand of silk magnified so that it looks like coarse canvas. The image quality is such that researchers examine these giant photos rather than use microscopes on the original.

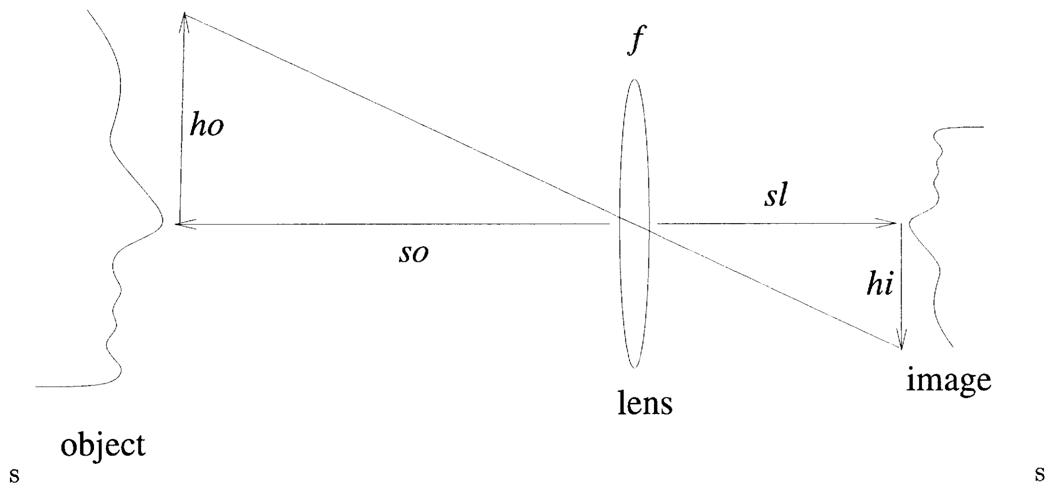


Figure 3-2: Height of imaged object on film.

From similar triangles we see that the magnification is just the ratio of the object and image distances:

$$frachiho = \frac{sl}{so} \tag{3.2}$$

We see from this that the variable focal length lens system always results in images at the same magnification, no matter where the focus, since both lens and film are fixed (so that sl and so remain constant). Only if the object itself moved would the scale of the image change.

Now let's say we have a fixed lens system. In such a system, we measure distances from the lens, since it is fixed: $d = so$. The image magnification for a fixed lens system is therefore

$$mag = \frac{sl}{d}$$

We would like to to see how the image scale changes for a single object imaged at two focal points, sl_1 and sl_2.

We'll take the ratio of the magnifications of each image:

$$ratio_{mag_1 \rightarrow mag_2} = \frac{mag_1}{mag_2} \tag{3.3}$$

$$= \frac{sl_1/d}{sl_2/d} \tag{3.4}$$
$$= \frac{sl_1}{sl_2} \tag{3.5}$$

We see that the distance the object is from the lens drops out of the equation, leaving only how far the film is from the lens. Therefore all the objects in image 1 will be at the same scale relative to one another as in image 2. Assuming both images were all in focus you could derive one image from the other merely by applying the scale ratio in eqn. 3.5

Doing the same test for a fixed film system we have an object imaged at two focal points sl_1 and sl_2. In a fixed film system we measure distances from the film: $d = sl + so$, since the film is fixed. The magnification ratio of that object between the two images is therefore:

$$ratio_{mag_1 \rightarrow mag_2} = \frac{mag_1}{mag_2} \tag{3.6}$$
$$= \frac{sl_1/(d - sl_1)}{sl_2/(d - sl_2)} \tag{3.7}$$
$$= \frac{sl_1(d - sl_2)}{sl_2(d - sl_1)} \tag{3.8}$$

In this system, d does not drop out, and you are left with relative scales from image 1 to image 2 which depend on d, as well as sl. Only very distant objects ($d \gg sl_1$, $d \gg sl_2$) will have the same relative scales. In fact this says that object in focus up close will be bigger relative to the distant objects than when the camera is focused on the distant objects. Even if both images were all in focus, you couldn't get from one image to the other by simply scaling them.

We conclude from this that the fixed lens system is to be preferred over the fixed film system, as it is easier to compensate images taken at different focal depths to be at the same scale. The variable focal length lens system is to be preferred most of all since all images are at the same scale and don't need compensation.

3.2 Image Scale Compensation

Remember that one of our objectives is to build a collage that is all in focus by pasting together all the in-focus parts of the original images. You can see such a collage in Figure A-5. Some of the source images that it was put together from are in Figures A-1.

In our system we've taken k images, $F_1[x, y]$ through $F_k[x, y]$ at k different focal points sl_1 through sl_k.

To make the collage we'll be taking pieces of each $F_i[x, y]$ and combining them into one frame. Which frame is used at each point is the frame most in focus at that point. Defining $m[x, y]$ to be the frame number most in focus at each point, we define the all-in-focus collage, $G[x, y]$:

$$G[x, y] = F_{m[x,y]}[x, y]$$

or simply:

$$G = F_{m[x,y]}$$

In order for this collage to be seamless and not have breaks in it, all the parts of F_i that get used in G have to be at the same scale. Otherwise parts of the collage taken from the different images won't line up with one another and you'll get half of one object small and the other half big. Therefore we'll need to insure that the parts that get used in G are at the same scale.

For the variable focal length lens this happens automatically as we've already determined. No matter where that system is focused, the images are at the same scale, since both lens and film positions are fixed.

For the fixed lens system we know the relative scale of the two images taken at sl_1 and sl_2 is sl_1/sl_2. So if we have a sequence of images F_i each taken at sl_i, we'll have to scale all but one of them to be at the same scale as the unscaled one. Let's say we want all images to be at the same scale as F_1.

To do this, images F_2 through F_k should be scaled by the inverse ratios of their magnification ratio with F_1. We'll call this scale X:

$$X_i = \frac{sl_1}{sl_i} \tag{3.9}$$

I wrote a Matlab program called **`flensmag(sl)`** that takes a vector of *sl* and returns a vector of X.

When I do this scaling, I just use a linear interpolator, since the scaling factors were all very close to 1. If space and speed permit you could do the scale optically with a zoom lens. See Willson and Shafer [11] for details.

Scale correcting for a fixed film system is harder since different parts of each image are at different relative scales (we've seen in the last section that the ratio of two magnifications depends both on sl and d). Thinking about it physically you can imagine the lens moving closer to the object as it moves farther away from the film in order to focus on the near part of the object. Since the lens moves closer to the object, we have so getting smaller and si getting larger. The near parts of the image are at a greater relative scale than the far parts when it is focused close like this. By moving the lens closer to the object you change not only the scale, but the relative scale of near to far, also known as perspective. Therefore no single scale factor will get *every* part of every image at the same relative scale. The best you can hope for is getting the *parts of the image that are in the collage* to be at the same scale. To do this you need to know which parts of each image are going to be in the collage.

Luckily we know that the only parts of each image F_i that should be in our collage G are the parts that are in focus. We can scale any two neighboring frames F_i and F_{i+1} assuming there exists a distance d such that both frames are in focus at that distance. We call this distance dav_i. To get the focused parts of F_{i+1} to be at the same scale as the focused parts of F_i you scale F_{i+1} by the inverse ratio of their magnifications:

$$X = \frac{dav_i - sl_i}{dav_i - sl_{i+1}}$$

To make *all* the images have their focused parts at the same scale you apply this scaling to successive pairs, resulting in a cumulative product of scales. You would scale each F_i by X_i, where X_i is the cumulative product of all the preceding scaling factors:

$$X_i = \prod_{j=2}^{i} \frac{dav_j - sl_j}{dav_j - sl_{j-1}} \tag{3.10}$$

X_1 is defined to be 1, so F_1 remains unscaled.

Now the parts of any pair of images that are both in focus (and therefore both in the collage) will be at the same magnification.

We have only to find good values for the sequence dav_i, somehow based on the sequence sl_i, the lens to film distances for each image. We could get dav_i by averaging the two distances that are completely in focus in each image. Remember that for the fixed lens: $d = si + so$ and $si = sl$ when the object at d is in focus, and $so = 1/(1/f - 1/si)$, so that:

$$d = sl + \frac{1}{\frac{1}{f} - \frac{1}{sl}}$$

We could average those:

$$dav_i = \frac{1}{2}(sl_i + \frac{1}{\frac{1}{f} - \frac{1}{sl_i}} + sl_{i+1} + \frac{1}{\frac{1}{f} - \frac{1}{sl_{i+1}}})$$

to get the dav_i at which to calculate X_i. The danger here is that one of those distances could be at infinity, making the average infinite as well. A better method is to average the two lens distances sl_i and sl_{i+1} and project that distance out:

$$\begin{aligned} slav_i &= \frac{1}{2}(sl_i + sl_{i+1}) \\ dav_i &= slav_i + \frac{1}{\frac{1}{f} - \frac{1}{slav_i}} \end{aligned}$$

This will give a more reasonable average between any two parts in focus since the amount of blurring (more on that next section) is more linear with sl then with so or d.

I wrote a matlab program called `mlensmag(sl)` that takes a vector of sl_i and returns a vector X_i. It does the average and scale calculation.

This program was applied to the source images in Figure A-1 to get the images in Figure A-2. You can easily see the difference. In the source images you can see more and more of the word "cheerios" as the focus changes from front to back and the magnification decreases. After compensation, the images in Figure A-2 all show the same amount of the word: "erios."

3.3 Blurring

Blurring caused by an image being out of focus can be modeled by convolution with the point spread function [17], or psf. The psf is a circle of unit area whose radius increases the more out of focus you are.

You can see why this is by tracing rays of light from a point source through the image plane and beyond, as we have done in Figure 3-3.

You can see that at sl away from the lens, the tightly focused point of light at si has spread into a circle of radius r_c. This radius is referred to as the "radius of confusion," and the circle is referred to as the "circle of confusion." The convolving function, h_c, is a circle of radius r_c and area 1.

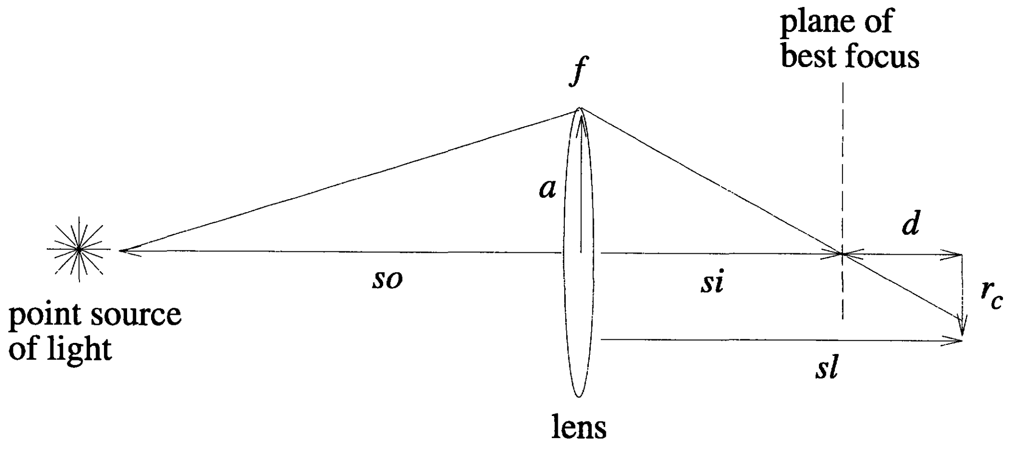


Figure 3-3: Relationship of distance from plane of best focus to radius of confusion.

$$h_c(r_c, x, y) = \begin{cases} \frac{1}{\pi r_c^2} & \text{for } x^2 + y^2 \leq r_c^2 \\ 0 & \text{otherwise} \end{cases}$$

This is an irascible function to convolve with. Its Fourier transform is a two dimensional bessel, which has multiple rings of zero magnitude. The sharp edges of the psf cause it to ring like that. Once convolved, those frequencies are lost forever, and the original image is difficult to recover; more difficult, say, than if you blurred the image by convolution with a gaussian, whose Fourier transform magnitude never completely hits zero. Deconvolving to reconstruct an image blurred by being out of focus is often done iteratively, since inverting the Fourier transform results in unbounded results at the zero crossings.

You can see from the figure that r_c is proportional to d, the distance the imaging device is from the plane of best focus, and proportional to a, the aperature. This explains why narrowing the aperature increases the depth of focus of an image: the circle of confusion stays quite small, and doesn't blur the image much.

3.4 Preserving Linearity of R_c in the Focus Parameter

We're going to skip ahead a little and talk about contrast curves and curve fitting. You may want to read the next two chapters on contrast curve modeling and curve fitting before coming back here.

Since the amount of blurring is a function of the absolute value of r_c, we would like to have our contrast curves plotted against an x axis linear in r_c. That way the contrast curve will be symmetrical and peak finding algorithms will not be biased by nonlinearity in that axis.

We'll examine the three optical systems we've already discussed, and find ways to map the variable used for focusing into a space linear in r_c.

First we solve for r_c given the system parameters sl, si, and a, as seen in Figure 3-3.

$$r_c(sl, si) = \frac{a\,sl}{si} - a \tag{3.11}$$

Our task now is to keep r_c linear in the focus parameter. The distance from the object to the camera is kept constant, since every contrast curve measures an object's distance from the camera. The lens aperature, a, is constant as well.

3.4.1 Variable Focal Length System

In this system, the lens and imaging device are both fixed, and focusing is done by varying the focal length f of the lens. We want to solve for r_c in terms of f. The distance of the object from the lens, so is constant. Combining Equations 3.1 and 3.11:

$$r_c(f) = a\,sl\left(\frac{1}{f} - \frac{1}{so}\right) - a$$

With a little algebra:

$$r_c(f) = a\,sl\frac{1}{f} - a\left(\frac{sl}{so} - 1\right)$$

We see that r_c is not linear in f at all but is linear with its inverse. When using this system what you'd like to do is use

$$x = \frac{1}{f}$$

as the x coordinate of the contrast curve. We will call this process *inversion compensation.*

When you find the peak of the contrast curve in x, just retransform it and solve for so to get the object distance from the lens. We know that at this peak, $sl = si$, because the object is in focus.

$$so(x) = \frac{sl}{x\,sl - 1}$$

3.4.2 Fixed Lens System

Looking at Equation 3.11 we see that the fixed lens system, which focuses by varying sl, already *is* linear in r_c. However, this linearity is destroyed by scale compensation. Recall that every image F_i is scaled by X_i, as shown in Equation 3.9. This scaling up process has the effect of making the *measured* value of r_c (measured by the contrast operator) seem that much larger. We'll call this measured value r_c'. Using Equation 3.9:

$$r_c'(sl) = r_c \frac{sl_1}{sl} \tag{3.12}$$

where sl_1, the imaging device to lens distance of frame 1, is a constant. Combining this with Equation 3.11 and simplifying:

$$r_c'(sl) = a\frac{sl_1}{si} - a\, sl_1 \frac{1}{sl} \tag{3.13}$$

Like the variable focal length lens, the effect of having all the pictures be at the same scale is to make the focal parameter, in this case sl, be inversely proportional to something linear in r_c. You want to use

$$x = 1/sl$$

as the x axis for the contrast curves. Then when you have the peak in that space, you invert it and solve for so:

$$so(x) = \frac{f}{1 - f\,x} \tag{3.14}$$

Another way to compensate is by changing the y coordinates of the curve, rather than the x. If we can figure out how scaling an image affects its contrast, we can compensate for it. We know that contrast operator just takes the first difference of pixel data and squares it. This is much like a gradient. Assuming that the color data is approximately linear between any two pixels, the gradient is inversely proportional to what we scale the image by. Scaling an image by factor k would therefore scale the contrast measure by a factor of $1/k^2$. We can compensate for that by multiplying the contrast measurement by k^2. Since we know k is X_i from Equation 3.9, we can define the compensated y_i to be:

$$y_i' = y_i \frac{sl_1}{sl_i} \tag{3.15}$$

We will call this operation *magnitude compensation.* You should use y' and proceed with peak finding as usual. Keep in mind that assuming that data is linear between two pixels is not always a good approximation. At an edge for instance, where we usually get our best data, it's closer to a step, which looks the same no matter how its scaled.

3.4.3 Fixed Film System

The fixed film system is amenable to both inversion and magnitude compensation, though they are both going to be only approximate. The problem is that the object distance d that remains constant throughout each point on the curve isn't $d = so$, like the others, but is $d = si + so$. Focusing in this system requires changing both si and so. This makes the scaling equations much more complicated. However, we will show that this doesn't matter so much, and that a typical fixed film system is very similar to a fixed lens system.

In the limit of ($d \gg sl_1$ and $d \gg sl_2$), Equation 3.8, defining scaling between two images in a fixed film system, becomes Equation 3.5, defining scaling between two images in a fixed lens system. Typically, d will be at least 10 times sl, so this is not a bad approximation.

Looking at it another way, the magnification factor of an image is equal to sl/so (Equation 3.2). In a fixed film system changing sl by some amount changes so by the negative of that amount. However, since so is typically at least 10 times sl you can ignore the change in so, and treat the two variables as independent, like in a fixed film system.

3.4.4 Experimental Verification

I tried both magnification and inversion compensation on experimental data collected at an edge. You can see the results in Figure 3-4. The top two pictures show the uncompensated curve. Notice that they lean noticeably to the left. The bottom two show the curves after each kind of compensation. The close-ups allow you to really see that the curves are more symmetrical after compensation.

To get an objective measure of just how much more symmetrical they were, I took the ratios of the heights of the curves at $\pm.5$ and ±1. Ideally the curves should be perfectly symmetrical about $x = 0$ and the ratios should both be 1.

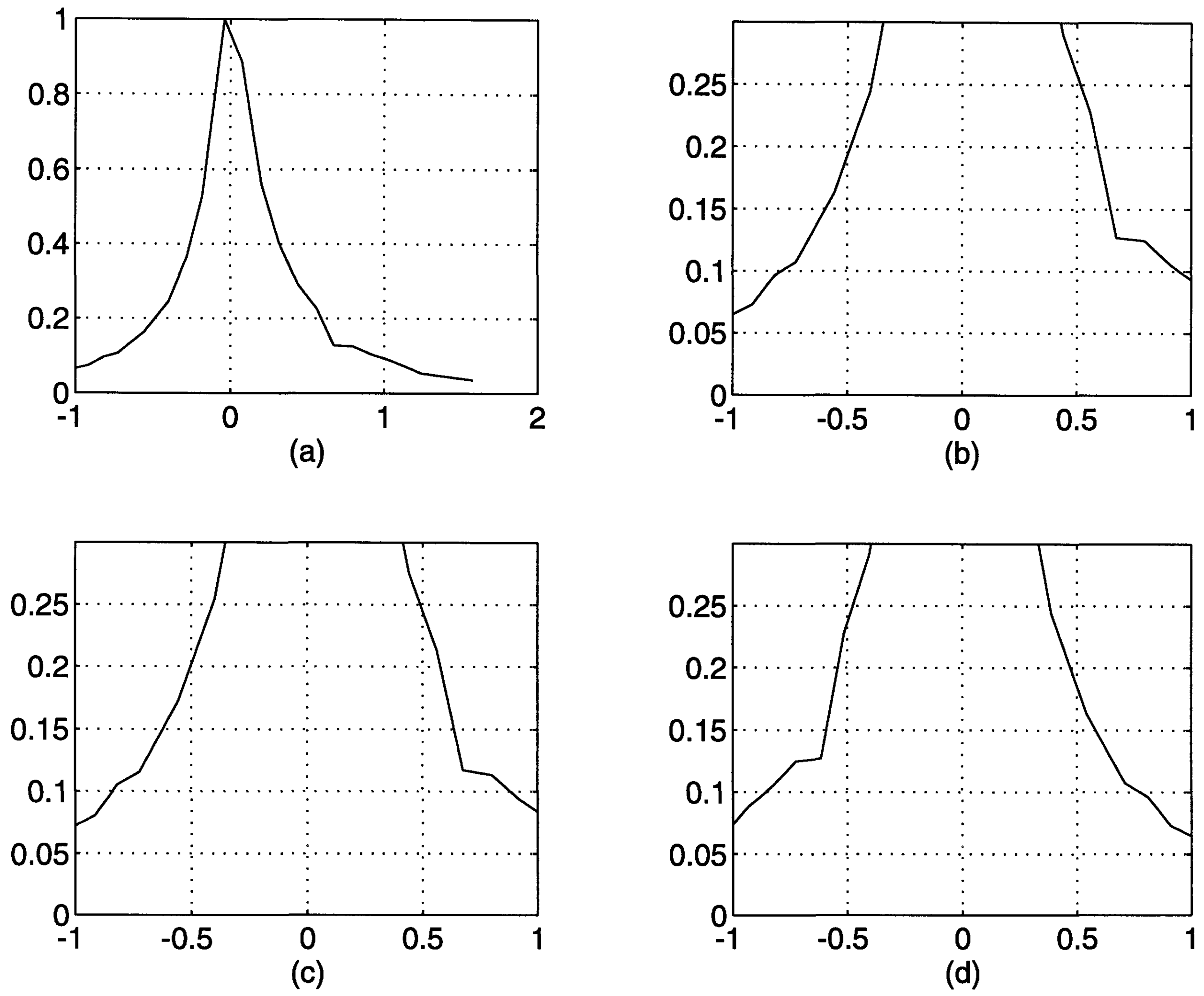


Figure 3-4: Measured contrast curves showing the effect of compensating *sl* in a fixed film system to preserve linearity in r_c. (a) Uncompensated curve. (b) Uncompensated close-up. (c) Magnitude compensation. (d) Inverse compensation. Note that (c) and (d) are more symmetrical about the y axis than (b).

Compensation Type	Ratio at $\pm.5$	Ratio at ±1
uncompensated	1.36	1.28
inversion compensation	1.25	1.15
magnitude compensation	1.23	1.12

You can see that both compensation types resulted in about a ten percent improvement in symmetry around both points. However, the curves are still noticeably asymmetrical. Fieguth [4] notices this asymmetry and shows that some occurs naturally and can be modeled using a thick lens model and ray tracing. We could be seeing that.

To see how necessary linearity compensation is, I applied it to the contrast curves, found the peaks using a straight line peak finder (more on that later), and mapped those into actual depths. The results of that experiment are in the following table.

Compensation Type	Distance in mm
uncompensated	1030.12
inversion compensation	1030.51
magnitude compensation	1032.16

Note that the inversion compensation results in a value only .4mm different from the uncompensated, a .04 percent difference. The magnitude compensation is more noticeably different, at 2mm. However, as we've already said, the linear data approximation that magnitude compensation rests on breaks down for sharp unblurred edges, which is exactly where the peak of the contrast curve is. Therefore I do not consider the magnitude compensation a good method to use when you're getting your contrast data from edges.

I think its clear that sl (or f for the variable focal length system) is very linear over the small range of points that you fit the peak finding curve to. The gross nonlinearity of sl (or f) only truly manifests itself over the distance of the whole curve.

Chapter 4

Model of Half Plane Step

In order to get an idea of what the contrast measure curves should look like we're going to develop a model of the curve at and around the center of a half plane step (otherwise known as an edge). This will aid is interpreting experimental data and help us in choosing curve fitting algorithms to find the peak of measured contrast curves.

We could have chosen a different model, one modeling a slope or a Markov style texture, but it was seen that edges provided very strong data and clear peaks. I do however recommend modeling these other kinds of curves since typical scenes are composed of a combination of edges, smoothly varying shading, and textures. Really accurate peak finding algorithms are going to probably want to choose a curve fitting algorithm based on the kind of curve it encounters.

When discussing images and algorithms, we will be distinguishing continuous 2d functions like $f(x, y)$ from discrete functions $f[i, j]$ by the use of brackets and parentheses.

We choose as our measure of contrast the Tenengrad operator with zero threshold, perhaps the most common of contrast measures. Nayar [16] settled on a sum-modified-laplacian which was clearly better suited to his rough, textured surfaces. No single contrast measure is best suited for all image types.

The Tenengrad operator is the sum squared magnitude of the first difference in the x and y directions (the discrete version of a gradient). It sometimes has a threshold value that forces the measure to zero if it is under that threshold. This is presumably for eliminating noise. We elmiminate this threshold as it unfairly penalizes edges for being darker than other edges even in the presence of little noise. Also the value of this threshold must be

$$\begin{tikzcd} f_s(x) \arrow[r, "\text{optical blurring by circle of confusion}"] & f_s'(x) \arrow[r, "\text{quantization by ccd array}"] & g(x) \arrow[r, "\text{sampling}"] & g[x] \end{tikzcd}$$

Figure 4-1: Process diagram of modeling contrast measure of half plane.

individually picked based on noise levels, average brightness of the scene, etc.

The Tenengrad contrast measuring operator is:

$$c_i[x,y] = (2f_i[x,y] - f_i[x+1,y] - f_i[x,y+1])^2 \tag{4.1}$$

summed over the three rgb color planes. It is usually summed over a window as well.

We would like to get some idea of what $c_i[x,y]$ should look like around good high contrast data like an edge. To do this we're going to model an edge with a half plane step, convolve it with a blurring function to simulate it being out of focus, convolve it with a function to simulate quantization by a ccd array, then sample and apply the Tenengrad operator to find its contrast curve. We see this process in Figure 4-1.

We start by defining the half plane step in 2d space:

$$f_s(x,y) = \begin{cases} 1 & \text{for } x \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

To simluate blurring we convolve it with a circle of confusion, with radius r_c, as described in the last chapter:

$$h_c(r_c, x, y) = \begin{cases} \frac{1}{\pi r_c^2} & \text{if } x^2 + y^2 \leq r_c^2 \\ 0 & \text{otherwise} \end{cases}$$

We call the result of the convolution f_s'

$$f_s' = f_s * h_c$$

Because the step $f_s(x,y)$ is actually completely independent of y, its convolution $f_s'(x,y)$ is as well and we need worry only about $f_s'(x)$.

Performing a convolution with a step and a circle is the same as determining the area of the circle over the vertical axis.

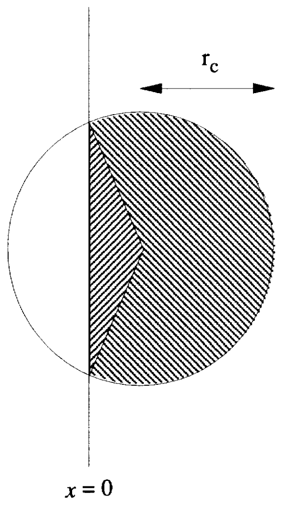


Figure 4-2: Convolution of circle and step is area of circle over the y axis.

When the circle is entirely to the right of the vertical axis the convolution result is 1, and when it is entirely to the left of the vertical axis it is zero.

$$f_s'(x) = \begin{cases} 1 & \text{for } x > r_c \\ 0 & \text{for } x < -r_c \end{cases}$$

In between these two extremes we see that only some of the circle is inside. We can break down the area of the circle into an arc and a triangle, shown in Figure 4-2, and sum them:

$$f_s'(x) = 1 + \frac{-\tan^{-1}(\sqrt{r_c^2 - x^2}/x)}{\pi} + \frac{x\sqrt{r_c^2 - x^2}}{\pi r_c^2} \tag{4.2}$$

This is only holds for $x > 0$. By symmetry and superposition we see:

$$f_s'(x) = 1 - f'(-x) \tag{4.3}$$

I wrote a Matlab function that returns this curve called `blurstep(x,r)`.

We plot $f_s'(x)$ for $r_c = 1$ in Figure 4-3, along with a linearization done in dotted lines. We see how close f_s' is to being linear, embarrasingly so. The linear model corresponds to convolving with a square rather than a circle. We'll call the linear model $f_l'(x)$:

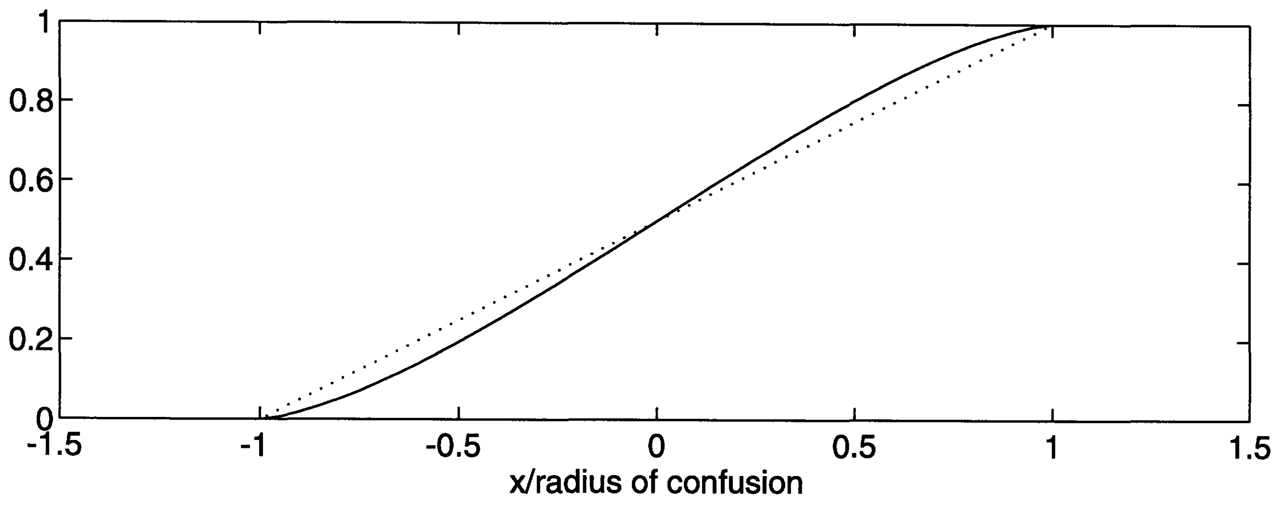


Figure 4-3: Plot of blurred half step. Linearization in dotted lines.

$$f_l'(x) = \begin{cases} 0 & \text{for } x < -r_c \\ 1 & \text{for } x > r_c \\ \frac{1+x}{2r_c} & \text{otherwise} \end{cases} \tag{4.4}$$

4.1 Contrast Measure of Blurred Step

Scaling r_c such that pixels are spaced 1 apart, the contrast measure at some point x is:

$$c[x] = (f'(x - .5) - f'(x + .5))^2$$

We'll pick one point to look at ($x = 0$) and vary r_c, so that we have

$$c[0] = (f'(-.5) - f'(.5))^2 \tag{4.5}$$

Combining 4.5 and 4.3:

$$c[0] = (1 - 2f'(.5))^2$$

combining with 4.2 yields:

$$c(r_c)[0] = (\frac{r_c^2 \tan^{-1}(\sqrt{r^2 - .25}/.5) - .5\sqrt{r^2 - .25}}{\pi r_c^2} - 1)^2$$

This is plotted along with its linear approximation in Figure 4-4. The two are not very different.

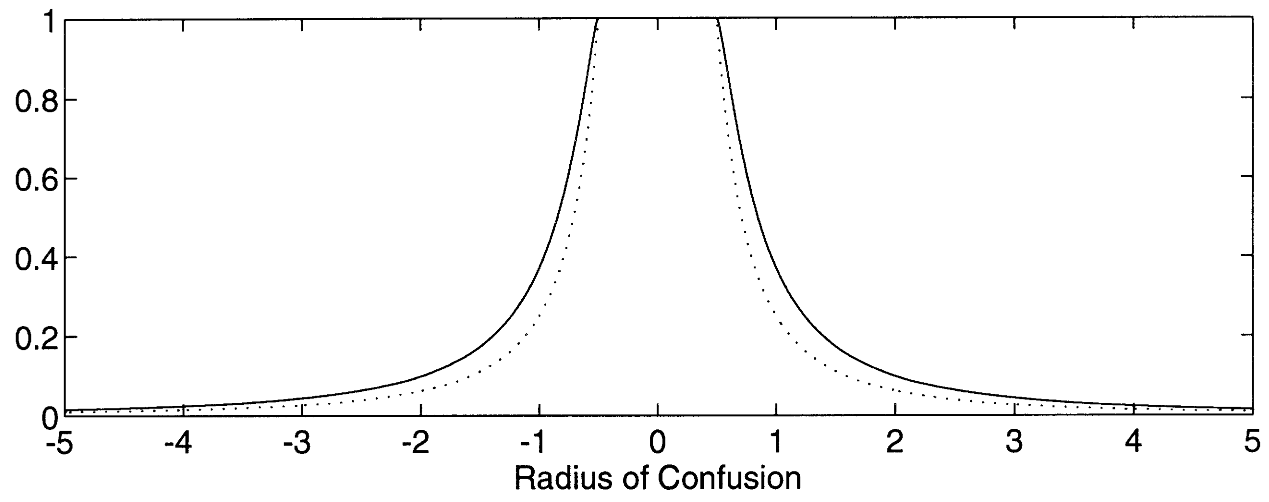


Figure 4-4: Plot of contrast measure of blurred step. Linear model in dotted lines.

The contrast evaluation of the linear model is not difficult. Combining Equations 4.4 and 4.1 we get the linear approximation of the contrast curve:

$$c_l(r_c)[0] = \begin{cases} 1 & \text{for } |r_c| \leq .5 \\ \frac{1}{4r_c^2} & \text{otherwise} \end{cases}$$

Looking at Figure 4-4, you can see that this is quite a tricky curve to find the center of. Unlike most curves one thinks of, you can only find its center by examining points *away* from its center. If you only had points taken at $r < .5$ you'd see only a flat line and be unable to find a center. You need data taken at $r > .5$, which is equivalent to the circle of confusion being bigger than your pixel size.

It looks like rather than using a parabolic fit about the top 3 or more points you'd be better off fitting $1/(4r_c^2)$ using points on the sides. However, these curves do not take into account the filtering aspects of the ccd array used to acquire the blurred image, which we will be examining next.

4.2 Effect of CCD Quantizing and Sampling on the Measure of Contrast

We can model a ccd array as being composed of tiny squares, each of which integrates the light falling across its surface over the time of exposure.

A typical way of approximating this is by convolution with a square the size of the pixel, and area of 1. In 2d, this blurring function h_{ccd} would be:

$$h_{ccd}(x,y) = \begin{cases} 1 & \text{for } |x| < .5 \text{ and } |y| < .5 \\ 0 & \text{otherwise} \end{cases}$$

Or since $f'(x)$ is in 1d:

$$h_{ccd}(x) = \begin{cases} 1 - .5x & \text{for } x < .5 \\ 0 & \text{otherwise} \end{cases}$$

More realistically, each photosensitive ccd square is somewhat smaller than pixel size [14], but this will give us the basic shape. Some sources model h_{ccd} as a gaussian, but really, engineers have a tendency to get gaussian happy, and it's better modeled as a square. We call the result of this convolution $g(x)$ and sample it at one pixel per unit to get $g[x]$.

$$g[x] = f'_s(x) * h_{ccd}(x)$$

The convolution of f'_s with h_{ccd} is left as an exercise to the reader. I do the convolution numerically in the program **fbcon(x,rc)** and the resulting contrast curve at the step edge is seen in Figure 4-5, along with the linear model in dotted lines.

Integrating the linear model:

$$g_l(x) = \int_{x-.5}^{x+.5} f'_l(x)dx$$

Is not difficult, the results at .5 and -.5 are presented here:

$$g_l(.5) = \begin{cases} 1/2 + 1/4r_c & \text{for } r_c > 1 \\ 1 - r_c/4 & \text{otherwise} \end{cases}$$

$$g_l(-.5) = \begin{cases} 1/2 - 1/4r_c & \text{for } r_c > 1 \\ r_c/4 & \text{otherwise} \end{cases}$$

We use these two points to get the contrast measure at the edge:

$$c_l[0] = (g_l(.5) - g_l(-.5))^2$$

which is

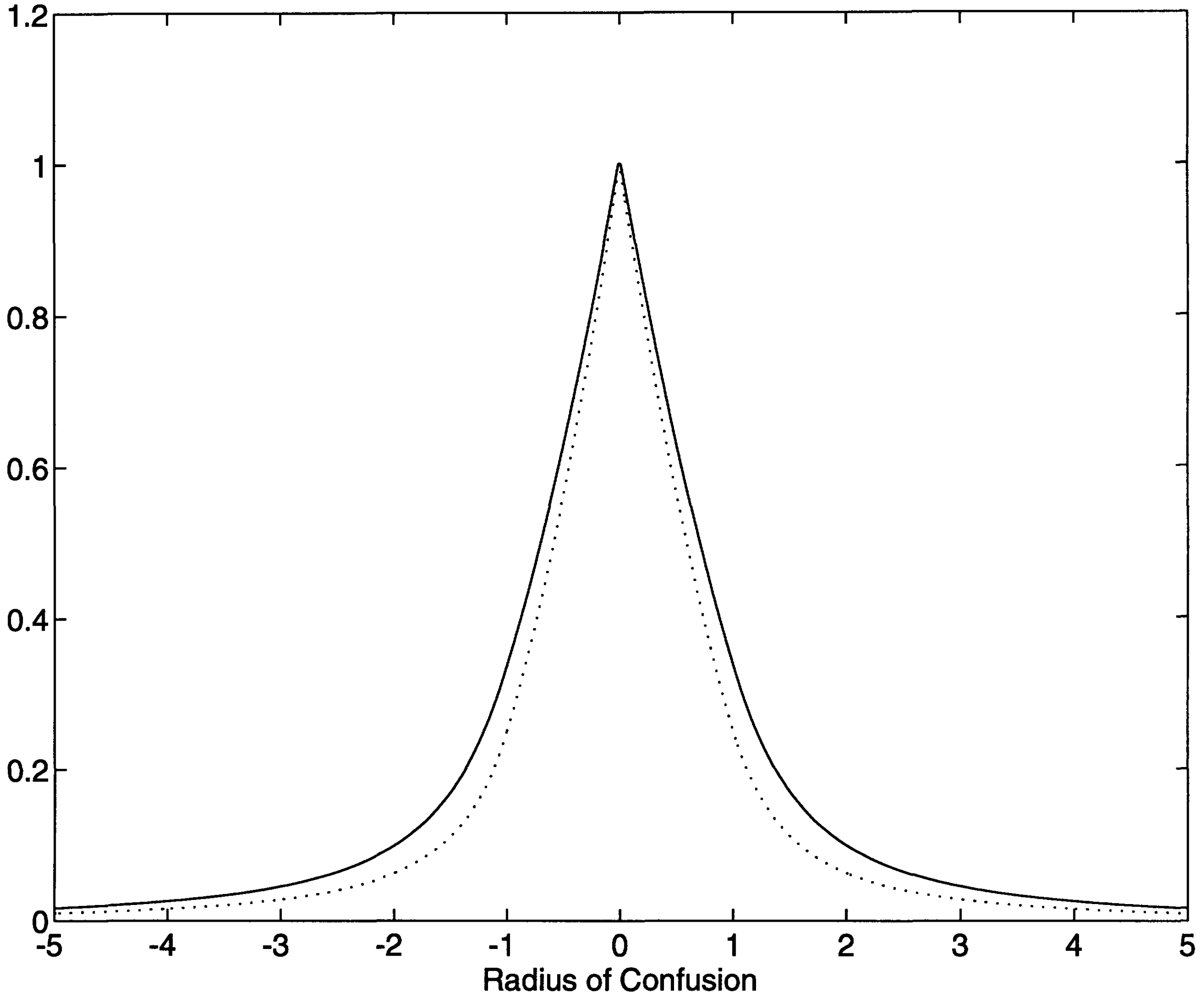


Figure 4-5: Contrast curve of $g(x)$, the result of further blurring f_s' with ccd array h_{ccd}. Linear model in dotted lines.

$$c_l(r_c)[0] = \begin{cases} 1/4r_c^2 & \text{for } r_c \geq 1 \\ 1 - r_c - r_c^2/4 & \text{otherwise} \end{cases}$$

We see this plotted in Figure 4-5 as the dotted lines. It's quite similar to the numerical integration of the non-linear model.

The linear model's break point occurs at $r_c = 1$ which is $g_l(.5) = .25$ You can see that the top might as well be linear, and the bottom is a $1/r^2$ curve.

We see that it is only because of the blurring caused by sampling with the ccd array that we can hope to find the peak of the contrast curve by looking at its very top. The flat peak of Figure 4-4 has been convolved into a triangular peak by h_{ccd}.

What happens when we look at the contrast curve away from the center of the step? I ran the numerical integration of Equation 4.2 at values of x away from the center. Four curves are shown in Figure 4-6. We see that away from the center of the edge the peak actually turns into a depression! The depresson occurs since when the image is completely in focus, there is only totally light or dark, then when it goes out of focus, the edge blurs into this otherwise undisturbed region, introducing contrast, that peaks when the radius of confusion equals the distance from the center of the edge.

Even one pixel away from the edge it hardly makes sense to look for the *peak* of the data, as it will simply find one of the two peaks away from the actual center of the curve. We'll be calling these fake peaks *side peaks* from now on. The way most sources deal with this problem is by summing the contrast over big windows, ideally windows big enough to bridge the maximum blank space between any two edges. The peak at the very edge is then big enough that it overrides the smaller side peaks.

This however has the effect of brutally smoothing out your depth data, leaving you unable to map the depth of anything with a sharply changing slope. Rather than use such a big window, it would be better to use a small window (I use 3x3) and use an error correcting algorithm to correct the data in the blanker regions. We discuss these issues more in the following two chapters.

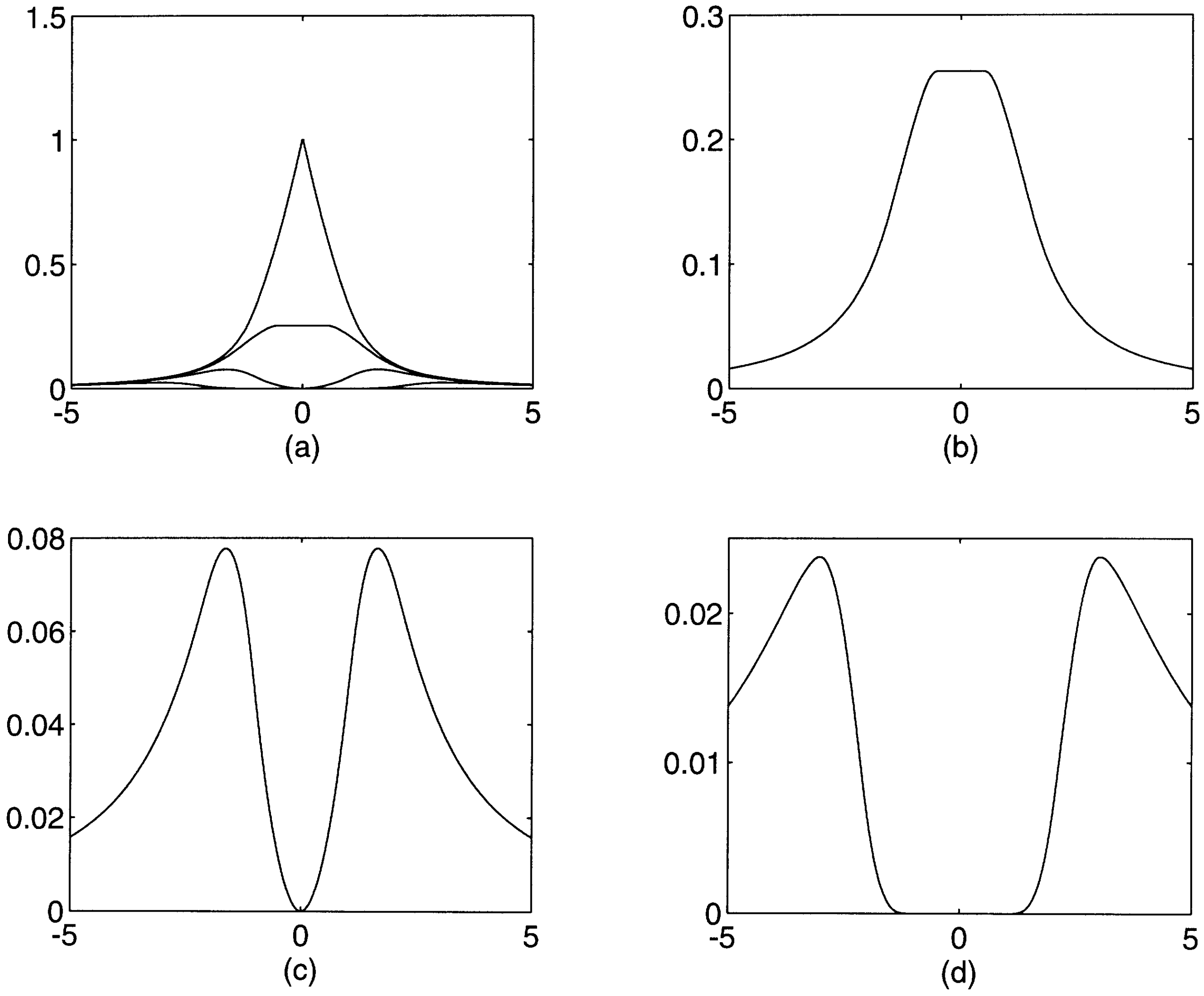


Figure 4-6: Plots of contrast curve away from the center of the step. (a) On the edge with (b),(c), and (d) superimposed. (b) .5 pixels from the edge. (c) 1 pixel from the edge. (d) 2 pixels from the edge.

4.3 Experimental Data

I scanned a half step using a standard 35mm camera and a scanner. You can see the contrast curves derived therefrom in Figure 4-7. One curve is taken from the very center of the edge, and the other two are taken two pixels to either side of the edge.

Figure 4-7(a) looks much like the model, and Figures 4-7(b) and (c) have the characteristic center dip that we see in Figure 4-6.

Notice, however, that Figure 4-7(c), you can see that in the white part, there is an upward spike right at what should be the bottom of the dip. This is because the white part of our imaged step is not perfectly blank, but has slight texture. When we know what we're looking at these things are quite easy to point out, but if we were just handed Figure 4-7(b), it might not be so easy to point to the small peak between the two other big peaks and say: "That one!"

We can assume that any real image is going to have such texture even in its smooth parts, and any curve fitting or peak finding algorithm is going to have to be able to deal with it somehow.

4.4 From Half Plane Step to Full Color Edges

Of course the half plane step does not exactly correspond to detail you're likely to find in an actual image.

For one, the positive part is likely to be different than 1.0, for instance it might be as much as 255 with 8 bit sampling, and that's just one color plane.

Any edge we find probably does not go just from black to white but could be from blue to red or green to brown. So that the color planes at the edge between any two arbitrary colors would be more like:

$$\begin{aligned} f_{red}(x,y) &= af_s(x,y)+b \\ f_{grn}(x,y) &= cf_s(x,y)+d \\ f_{blu}(x,y) &= ef_s(x,y)+f \end{aligned}$$

Which should worry us since our method of finding contrast is to add up the individual contrasts from each color plane. What sort of mess will that result in? Convolving red with

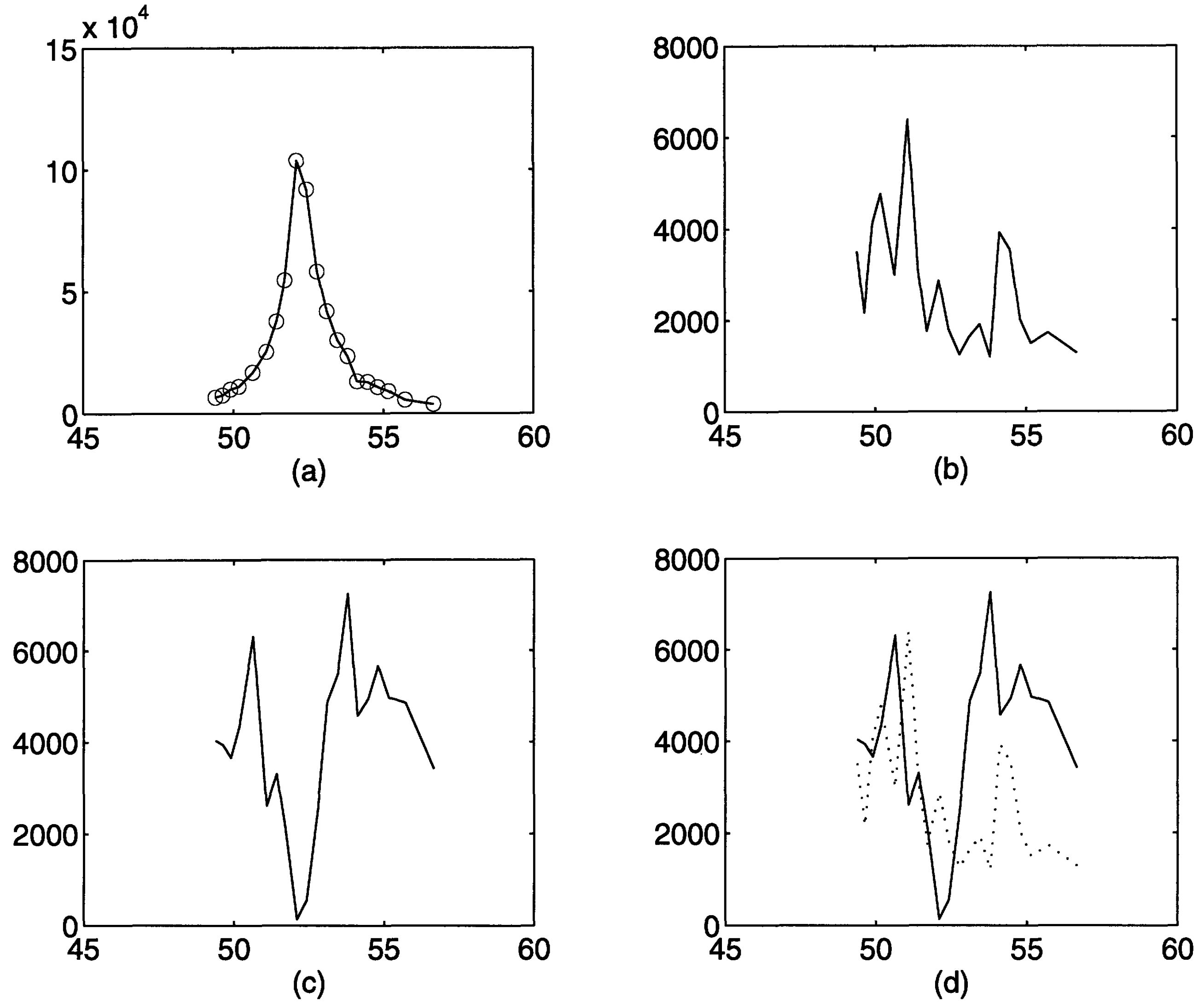


Figure 4-7: Experimentally obtained contrast curves. (a) From the center. (b) From the white part of the step. (c) From the black part of the step. (d) Black and white curves superimposed.

the circle of confusion h_c

$$f'_{red}(x) = f_{red}(x, y) * h_c(x, y)$$

since convolution is linear, and the area under h_c is 1

$$\begin{aligned} f'_{red}(x) &= a(h_c * f_s) + b \\ f'_{grn}(x) &= c(h_c * f_s) + d \\ f'_{blu}(x) &= e(h_c * f_s) + f \end{aligned}$$

since $h_c * f_s$ is f'_s

$$\begin{aligned} f'_{red}(x) &= af'_s + b \\ f'_{grn}(x) &= cf'_s + d \\ f'_{blu}(x) &= ef'_s + f \end{aligned}$$

Further convolving f' with the pixel blurring function h_{ccd} whose area is also 1

$$\begin{aligned} g_{red}[x] &= ag[x] + b \\ g_{grn}[x] &= cg[x] + d \\ g_{blu}[x] &= eg[x] + f \end{aligned}$$

Our contrast measure is the first difference, squared, summed over the three planes

$$\begin{aligned} c_{red}[x] &= (g_{red}[x + .5] - g_{red}[x - .5])^2 \\ &= a^2(g[x + .5] - g[x - .5])^2 \end{aligned}$$

We then add up all three planes for the total contrast measure.

$$\begin{aligned} c_{tot}[x] &= c_{red}[x] + c_{grn}[x] + c_{blu}[x] \\ &= (a^2 + c^2 + e^2)c[x] \end{aligned}$$

Defining a new variable $k = a^2 + c^2 + e^2$, we see that in fact the contrast curve of an edge between any two colors is just a scaled version of the unit step response contrast curve:

$$c_{tot}[x] = kc[x]$$

The other way we could have combined the three planes is to sum the planes before taking the first difference and squaring it, so that

$$g_{tot}[x] = g_{red}[x] + g_{grn}[x] + g_{blu}[x]$$

In which case we'd still have a scaled curve:

$$c_{tot}[x] = kc_s[x]$$

except that k in this case would be $(a + c + e)^2$ which I consider dangerous as it can cancel itself out whereas $a^2 + c^2 + e^2$ can never be zero unless all three variables are zero. The advantage of doing it like that would be lower noise in the final contrast measure. Squaring the first difference couples additive white noise with the signal, and adding the pixels together first would lower the noise level that gets coupled.

We see that for an edge between any two colors we still have the original step response contrast curve, only with a scale factor. We see thereby that any curve fitting algorithm we use to find the peak of the curve given a few points on it will have to be scale invariant in y. However, it does not have to be offset invariant. By taking the gradient of the image we cancel any DC bias there might be.

Chapter 5

Different Curve Fits Based on the Model

Now that we have a model we can get experimental data and try to apply the model to that data. To get an actual contrast curve we apply the Tenengrad operator (Equation 4.1) to our images $F_1[x,y]$ through $F_k[x,y]$. We get from that contrast curves in three dimensions $c_i[x,y]$, where every (x,y) point pair in the scene has a curve indexed by i. From here on we'll assume the (x,y) location is given and talk only about $c[i]$.

We expect this curve (at least in cases close to edges) to look alot like c_s, the curve we derived in the last section.

Each curve has k points, where k is the number of images we took. Each value of $c[i]$ has a corresponding $sl[i]$, the compensated lens distance from the film at frame F_i. You need to compensate sl according to one of the algorithms discussed in the Optics chaper so that r_c will be more nearly linear in sl. Our curve is the function defined by k pairs of x and y: $(sl[1], c[1])$ through $(sl[k], c[k])$.

Our task then is to find the peak of this curve, the value of sl where c would be at its maximum. This value of sl is where the radius of confusion is zero, the distance of the image plane of best focus from the lens, which we can then use to find the distance of the object.

For the rest of this chapter we'll be talking about x_i and y_i rather than $sl[i]$ and $c[i]$, which is an easier notation to look at for curve fitting. Assume that y_2 is the maximum of the contrast curve, and points (x_1, y_1) and (x_3, y_3) are the two points immediately around it.

Our task is to interpolate a curve through the points and find the value of x that the interpolated curve peaks at, this value being called x_p, for peak.

5.1 Parabolic Fit

I'm going to go through a parabolic fit first simply because it is the most commonly used, though cubic and gaussian are also sometimes used.

The parabolic fit is:

$$x_p = \frac{1}{2}\frac{x_1^2(y_2 - y_3) + x_2^2(y_3 - y_1) + x_3^2(y_1 - y_2)}{x_1(y_2 - y_3) + x_2(y_3 - y_1) + x_3(y_1 - y_2)}$$

Which requires 6 multiplies, 7 additions, and 1 division.

This is a reasonably well behaved curve fit. It is scale invariant in y, meaning that if you scale y, x_p remains constant, which is what we want. Its major problem is that it doesn't represent the curve well, at least theoretically. When used in literature, it is only used as a fit to the very top of the curve, where it does look parabolic but which requires too many points of data for a real time system.

A more appropriate variant might be a gaussian fit, since, like the model we have, it asymptotically approaches zero at its ends rather than continuing negative like a parabola would. You can fit the gaussian by doing a parabolic fit on the log of the data. The gaussian might also have nice properties for error correction, since by taking the log of the data, the curve's scale is reduced to a shift, and the measure of the gaussian's curvature (which might be taken as an indication of how good the underlying curve data is) is therefore scale invariant.

Taking a log doesn't have to be slow, it can be largely table driven, $\log_2$ is particularly easy to implement via a combination of most significant bit detection and table lookup.

Experimentally, I found the parabolic fit to suffer from over-sensitivity to the maximal point. It unfairly biases the location of the peak towards the largest value, y_2, so that $x_p = x_2 + \epsilon$. This happens because the curve we're trying to interpolate with a parabola is actually alot pointier than the parabola. The interpolated curve stair-steps, with the peaks jumping suddenly from one peak to the next, without smoothly varying between. I verified this phenomena on our model and you can see the stair stepping of the parabolic fit in Figure 5-1.

Using a gaussian fit mitigates this somewhat, but it's still apparent. The problem is that around solid contrast data with small windows, the contrast curve looks like the model, which is much pointier than a gaussian or parabola.

The Matlab program `parpeak(x,y)` does a parabolic interpolation, returning peak and curviness.

5.2 Inverse R^2 Fit

It might be advantageous to consider again the model we developed for the contrast curve. The curve decays at a rate of $1/r^2$ away from the peak. You could normalize the measured curve so that its peak was $y = 1$, and fit the $1/r^2$ curve to points under $y = .25$.

The problem with using this fit to find the center of the curve is that you have to look at points *away* from the very peak of the curve, where the data gets small in value compared to the peak and is more susceptible to noise.

To do it, you probably need to take many points, or all, around the peak, starting at about .25 under the maximal value of the curve.

Unlike a parabola, the $1/r^2$ curve only has two parameters, and therefore only needs two points minimally.

The equation defining the curve is:

$$y = \frac{k}{(x - x_p)^2}$$

where, as before, x_p is the location of the peak. Given two points on opposite sides of the peak, we can solve for x_p:

$$x_p = \frac{(y_2x_2 - y_1x_1) - |x_1 - x_2|\sqrt{y_1y_2}}{y_2 - y1}$$

Notice that this is scale invariant in y as well, but does require a square root. The square root is used as a logarythmic average of y_1 and y_2. If y_1 and y_2 are close enough in value (which they might be, since you take them both when the data is low,) you might be able to get away with simply averaging them, which would eliminate the need for a square root.

Experimentally, when fitting only two points I found this to be very susceptible to noise, and to shift the peak away from where it obviously is. A fit with more than two points might mitigate this, but only in the presence of very clean data.

The Matlab program `r2peak(x,y)` does this interpolation.

5.3 Inverse R Fit

An almost equally good approximation for the curve away from the peak may be $1/|r|$ which still dies off as it leaves the center, and is easier to solve for:

$$x_p = \frac{y_1 x_1 + y_2 x_2}{y_1 + y_2}$$

It too, is scale invariant in y, and suffers from noise sensitivity. The matlab program `r1peak(x,y)` does this interpolation. I could found neither one of these effective in finding the peak.

5.4 Two Straight Lines

Going back to our model you can see that the top of the curve in Figure 4-5 looks like nothing more than a triangle. It's like a parabola, only first order:

$$y = a|x - x_p| + k$$

You can take the maximum point, and two points around it, draw a line from the center point to the point around it that would give it the greatest slope, then draw a line from the other point up with that same slope and find the center.

Given the three points, and assuming the line from x_1 to x_2 has a greater slope than x_3 to x_2, the peak finding equation is:

$$x_p = \frac{x_1(y_3 - y_2) + x_2(y_1 - y_3) + x_3(y_1 - y_2)}{2(y_1 - y_2)}$$

The Matlab program that does this is `slpeak(x,y)`. Theoretically and experimentally I find this to be a superior curve fit over the parabolic or gaussian, at least when points on your curve are taken about 1 r_c apart, which ours are. If you had points closer together, you might want to do a parabolic. The primary feature of this straight line model is that it doesn't stair step as much, which you can see in Figure 5-1. This means that the depth map will better interpolate between values of sl, and not look as though it was quantized to k levels.

When running this stair step simulation I found that the straight line interpolation stair stepped just as badly as the others when the radius of confusion was allowed to grow greater than one between successive frames. Looking at Figure 4-5 you can see why that is. At r_c greater than one, you leave the linear looking region and enter the $1/r^2$ region. At that point you'd have to do a $1/r^2$ fit to avoid stair stepping. Doing a parabolic or gaussian fit you want the points even closer together, like you want to keep r_c under .3 or so. There are three ways to achieve this. One is by taking more frames. That will increase your overall accuracy but also takes longer, linear with how many frames you take. Another is by limiting your total range of focus, but this may be unsatisfactory. The other, which may be most desirable is by narrowing the aperature. That will narrow r_c proportionally.

You can actually calculate pretty accurately how big your maximum r_c difference between frames is. All you need to know is the size and pixel density of your imaging device, sl_i, and the effective aperature size.

5.5 Center of Symmetry

One thing all the curves shown in the model have in common is their symmetry about the center of best focus. Even the ones with 2 side peaks and a depressed middle have this symmetry. We can imagine then a peak finder based not on finding the center of any peak or depression, but instead simply finding the center of best symmetry about any curve.

I tried a center of mass algorithm.

$$x_p = \frac{\sum_{i=1}^{k} x_i y_i}{\sum_{i=1}^{k} x_i}$$

This worked well on nice looking curves with a strong central peak and data dying out quickly on either side, but failed on the interesting data where there is only one side peak on the curve, as it simply finds that false peak and not the middle. It will find the center between two peaks, which is nice.

A better method might be to treat the curve as a continuous function, interpolating with a linear or sinc and find the point x_p on curve $y(x)$ that minimizes the difference of the data around that peak.

$$x_p = \min \int (y(x_p + x) - y(x_p - x))\, dx$$

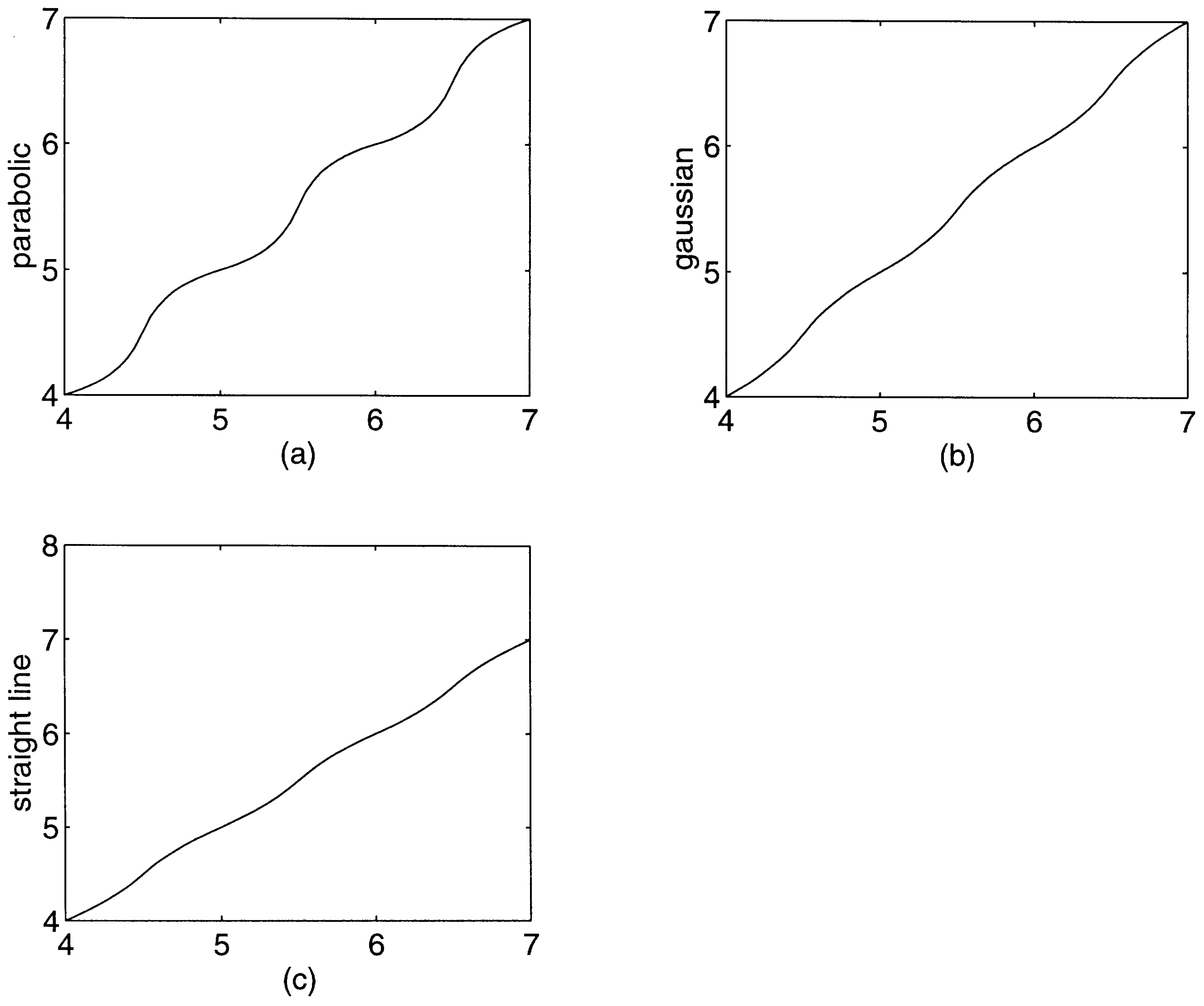


Figure 5-1: (a) Stair stepping effect of interpolating depth with parabolic fit. (b) Better fit using gaussian. (c) Best fit using straight lines.

The two problems here are the computational complexity and the fact that the point of symmetry might be sufficiently close to the edge of the curve that there isn't enough data to one side of it to say how well it matches the other side. The only way to avoid this is to increase the number and range of your acquired images such that the maximum radius of confusion in any of the images is greater than the distance any low contrast or blank area in the scene is from a strong edge (all distances in pixels, of course) This would guarantee enough data to work with to find a good point of symmetry, but unfortunately might not be physically possible. Neural networks might be good at this sort of thing, finding results from incomplete data, and shape detection.

Chapter 6

Error Correcting

Many error correcting schemes treat all points equally, and assume that one data point is as valid as another. Low pass filtering to reduce noise for instance, makes no distinctions between points. Other error correcting techniques like the noise reduction technique for reducing salt-and-pepper noise decide if a data point is valid by comparing it to its neighbors.

When correcting a depth map, we have the advantage of having more than just a single depth value to work with. We have access to the source of this data, the contrast curve, and should be able to assign some degree of probability to every point based on the shape of this curve. Once we have assigned some measure of probability to each point we can use some sort of interpolation scheme.

We will be calling the algorithm that measures this probability the *error metric.*

6.1 Finding Bad Data

Before examining a way of assigning probability to each curve fit in our depth map, we should try to see if we can identify points that are absolutely wrong, points for which the peak finding algorithms described in the last chapter will always fail.

These peak finding algorithms get hit by contrast curves taken slightly away from the center of an edge, modeled in Figure 4-6, and seen in Figure 4-7. In these cases there occur two peaks, rather than one, each of which is about as far from the actual center of best focus on the curve as the point is from the edge. Peak finding algorithms will simply take the bigger of the two peaks and find its center.

It really becomes bad when the center of best focus is close to one of the edges of the curve, so that only *one* of the side peaks actually makes it onto the curve. This can look an awful lot like a central peak. Figure 6-1 shows this well. By itself you would think that Figure 6-1(a) was a central peak, not a side peak.

At its most extreme, when really far from an edge, the whole curve is very low valued and only starts curling up at one or both ends of the curve. This case can be easily detected by discarding curves that have their maximum at an end. Of course, you must also ensure that no part of the scene actually is in best focus at either end of the curve, or good data will be lost.

One way to prevent this whole class of phenomena whereby distant edges create false peaks in the curve is to use a contrast summation window bigger than the biggest radius of confusion. That way, the relatively low valued side peaks are crushed by the strong central peak at the center of the edge and curve fitting algorithms find only the central peak. This is the standard approach, but suffers because it means that any quick changes in depth over the size of that window are averaged and smoothed over.

It would be preferable to identify the pathological cases, and discard all peak data from them, or even better, find the center of best focus by fitting to the depression. Experimentally, I've found this difficult since the depression is often very wide, and, in the presence of some small amount of local texture, may even have its own peak on the bottom of the depression, like in Figure 4-7(b).

I tried three algorithms, each designed to detect one of the three cases mentioned above.

What I did was to keep two maps around. One was the depth map, created from curve fitting, and the other was the probability map, created with the error metric. Data points that were absolutely wrong were assigned a probability of zero.

I easily eliminated the last case I mentioned, the case where most of the curve is flat, and only tips upwards at the very edges of the curves, by discarding all curves whose maximums were at their edges.

I also tried a "twin peak[1]" detecting algorithm, that looks for two peaks of any size. It is essentially a test for monotonicity. This was too extreme and threw out curves that had tiny secondary peaks due to noise or the presence of a second edge far away. To fine tune it a bit I refined the algorithm to only discard the points whose second peak was within a factor of 8 in magnitude to the main one. This worked alot better and you can see the

[1]This algorithm does *not* in fact detect the presence of David Lynch.

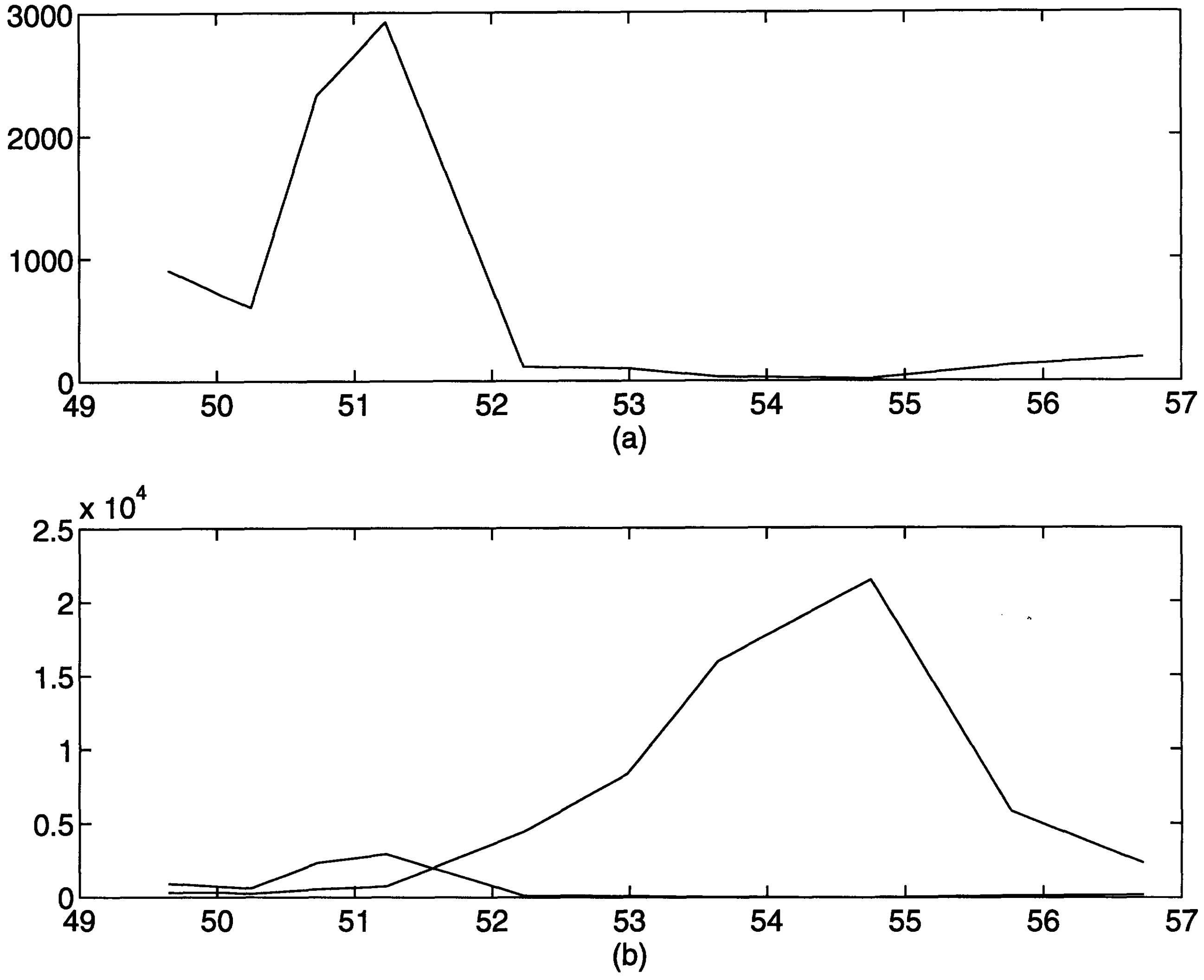


Figure 6-1: A false peak hard to find out of context. (a) The peak. (b) The peak superimposed on good data four pixels over, showing it up as a side peak.

difference between it and doing no twin peak detection in Figures A-3(a) and (b), most noticeably around the text on the game boxes.

I also tried an algorithm designed to detect the previously mentioned very deceptive single side peak of Figure 6-1. It finds the maximum of the curve (the side peak), the minimum (presumably the actual center of best focus), and then going in that direction looks at data at the end of the curve and sees if it has a positive slope going off the curve. This should work because the real central peak curves will be sloping downward as they go off the curve. Experimentally however, this algorithm was shown to be too enthusiastic in throwing out data points. A little noise or a distant edge can give even a good curve a slight upward slope at the end, even though the peak of the curve is right where it should be. I tried some fine tuning, but it didn't seem worth it.

Another method to consider is some combination of contrast and direct image processing. For instance, you can see from Figure 6-1 that the secondary peak is a factor of 10 smaller than the primary peak. You might want to keep some sort of local maximum that you can compare peaks to. Peaks falling greatly under this maximum should come under suspicion. The window over which the maximum was kept would have to be as big as the maximum radius of confusion in the images.

I did not implement a simple magnitude thresholding of the curve to eliminate noisy signals. I found that even when the contrast measure was very low, (powers of ten under the maximum in the scene) there were often small peaks present caused by local texture.

The program `isbad(x,y)` implements the bad data detecting algorithms that worked for me.

In general, I believe curve shape analysis has a lot of potential for distinguishing bad curves from good ones, or even better, allowing good data to be extracted from curves like the twin peaks curve.

6.2 Error Metrics

In this section we examine some algorithms that do not assign an absolute "good" or "bad" rating to the data but instead grade them on a continuous scale. The better the data, the higher the value should be. To compare these, I wrote a program that took only the best half of the data according to the metric and zeroed out all the other depth information. I then wrote these as color images. That way you can see which points each metric preferred over the other metrics. These images are all in Figure A-3.

6.2.1 Parabolic Curvature

My first inclination was to take the absolute value of the curvature coefficient from the parabolic curve fit. It is the value a in the following parabola equation:

$$y = a(x-p)^2 + k \tag{6.1}$$

The greater a is, the more "peaky" the data is. You could imagine that where there is no data at all, you'd have a flat curve, and a would be zero. Given three data points on the parabola, we can solve for a:

$$a = -\frac{y_1(x_2 - x_3) + y_2(x_3 - x_1) + y_3(x_1 - x_2)}{(x_1 - x_3)(x_2 - x_3)(x_1 - x_2)}$$

You can see from this that a is not scale invariant in y, in fact it is linear in y. Nonetheless I've kept this as my preferred metric for quite a long time. It works quite well, favoring all the really strong data in and around edges. Figure A-3(b) shows the thresholded depth map. You can see most of the points it kept are quite reasonable.

6.2.2 Normalized Curvature

I was disturbed however by the scale invariance of a in y. I decided to normalize a to the magnitude of the curve by dividing it by the maximum y value in the curve.

$$a' = \frac{a}{\max(y_i)}$$

Experimentally, the result of normalizing in this fashion was not too different from not normalizing. The main difference was that low signal strength false noise peaks were getting included as good data. You can see this on the walls in Figure A-3(c), which uses this metric.

I eventually decided that curvature is probably a lousy measure of data accuracy. Remember that the model predicts that all the curves will have the *same* shape, and hence the same amount of curvature. Most of the bad curves are the deceptive side peak ones which often have a *greater* curvature when normalized than the central peaks.

6.2.3 Magnitude of Peak

When I became suspicious of the scale variance of the a parameter, I decided to see just how dependent that metric was on magnitude by simply using the value of k derived from Equation 6.1, which is the peak of the parabola. The result is remarkably similar to using the curvature except that low level signals, both good and bad, are indiscriminately discarded, which you can see on the back wall in Figure A-3(d).

All these metrics of peakiness or intensity are simply not suitable. Neither can distinguish a side peak caused by a far away edge, and they either throw out lower level yet good data, or accept noise spikes as good central peaks.

If you are using some sort of symmetry detecting peak finder, you could use a measure of just how symmetrical the curve is as a measure of how accurate it is[2].

6.2.4 Fitting an Ideal Curve

Finally I decided that the only way to reliably grade the shape of a curve was to compare it to the ideal curve. This is where the $1/r^2$ curve in the model finally comes in handy, not as a peak finder, but as an error metric. What I did was to fit the $1/r^2$ to the data away from the peak and take a summed squared error between it and the data as a measure of how ideal the curve is. This algorithm is implemented in `fitpeak(x,y)`.

All by itself it is able to detect and discard the twin peak curves without any other algorithm. You can see that by comparing Figures A-3(e) and (f): (e) doesn't do twin peak finding, (f) does, but you can't see the difference. It is similar to the normalized curvature metric in that it does better finding good low level signals than the parabolic curvature metric, but is worse around strong signal levels in the foreground. Like every other metric we've tried, it can't distinguish single side peaks from central peaks.

6.3 Error Correcting Algorithms

We now explore ways of applying an error metric to reconstructing or correcting depth data in areas with poor accuracy. We go from a slow, "ideal" solution to faster more implementable ones, and rate their performance on actual data.

[2]Of course, that won't always work so well either. For instance, a flat line is perfectly symmetrical ... anywhere on it!

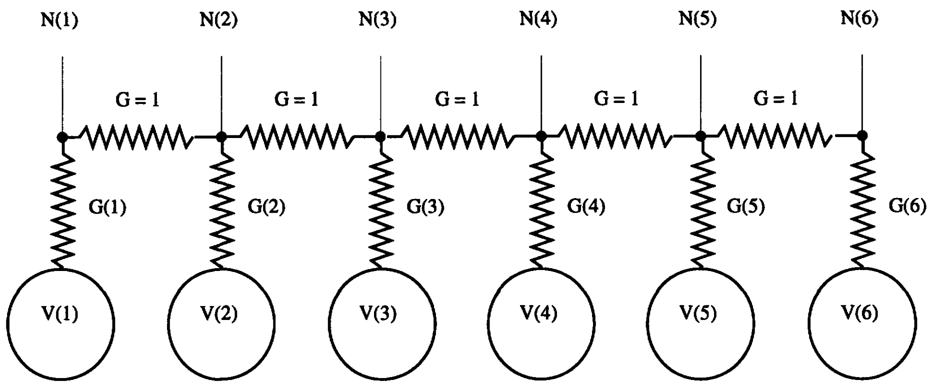


Figure 6-2: Slice of error correcting two dimensional resistor network.

6.3.1 Resistor Network

The "ideal" way of error correcting might be similar to the following system. Imagine a circuit with nodes corresponding to the points in the depth and probability maps. Every point in the depth map $d(x, y)$ is represented by a voltage source of voltage $V = d(x, y)$. Every point in the probability map $p(x, y)$ is represented by a resistor of conductance $G = p(x, y)$ connected at one end to the voltage source at the same point. Finally, the free ends of these resistors are connected to their four immediate neighbors by resistors of conductance 1. Figure 6-2 shows a one-dimensional slice of this network.

The corrected data will then simply appear on the output, $N(x, y)$. You can see how this works. Data points with a very high probability and hence conductance will simply connect N to V, whereas data points with low probability will be mostly cut off from their V and will be linearly interpolated by the resistors connecting neighboring nodes that have better data.

You can simulate this system in software, but it requires the solution of a tridiagonal matrix each side of which is as big as the product of the width and height of the depth map. It's terrifically slow, and takes hours even on a high end workstation. To increase speed you could do it in subcells or via a travelling window. You could also try iterative solutions or successive one-dimensional approximations.

Even so, its not suited to real-time solution at the moment. It also has difficulties regarding the scaling of the probability map. Not only do you need to get a linear scaling correct, you'll probably need to introduce a nonlinear mapping of some sort from probability map to conductance. Otherwise, you could end up with a blurred out mess.

Mostly this system gives us the idea that we should bolster up poor data by interpolating good data around it. The next few sections explore methods that should be quite fast, implementable in hardware, and perform reasonably well.

6.3.2 Zero Order Hold

Since the time required to solve that network is so gruesome we are going to explore methods that don't require big matrix solutions, but instead threshold the probability into "correct" data and "incorrect" data. There is some evidence to show that this is not wholly inappropriate. Large blank areas with little contrast are likely to be "incorrect" and areas with texture or near an edge are likely to simply be "correct".

In my images which have a lot of blank space, a threshold that included 50 percent of the pixels seemed about right.

Pictures with more detail can probably get away with a higher threshold.

What you want, ideally, is to stretch an elastic membrane across the data, pinning it to the good data, and letting it stretch over the bad data. You could solve for that system ideally, but again, it's slow, so we need an approximation.

The first approximation to try is a zero order hold, duplicating the last good value at each place that is not good. The Matlab programs `x0cor` and `x0corl` do this, and the result of this error correction is seen in Figure A-4(b).

This isn't bad for slowly varying depth data but becomes obviously choppy when applied to any depth data with a slope. This zero order hold would be very easily implemented in hardware, operating on an incoming stream of pixels and spitting them out again.

You could do the interpolation in the y direction given 2 frame buffers as big as the image. You would fill buffer 1, then fill buffer 2 as you scanned buffer 1 in the y direction, then switch again.

6.3.3 Linear Interpolation

A linear interpolation might be more in order, linearly interpolating the bad data between the last good depth data to the next good depth data.

The Matlab programs `x1cor` and `x1corl` do this. Figure A-4(c) shows it.

This interpolation is still not unfeasable in hardware, though slower than the zeroth order hold, and you can no longer process the incoming streat at the same speed. Instead you will have to process pixels much faster than they arrive in order to keep up since you need to hold a bunch of bad points until the next good point comes along.

6.3.4 Minimum Distance Linear X and Y

Only interpolating in the x direction is wasteful, there might be closer data in the y direction. Therefore, you should interpolate in both x and y, and somehow combine the two.

One method I used was to keep a distance map of both the x and y corrected images, where the distance map corresponded to how many pixels away the bad data was from the good data. Good data has a distance of zero. The next one over, if it is bad is assigned a distance of 2, then 3, until it reaches halfway to the next good data and counts down.

Given these distance maps you can then combine the two sets of 1d corrected data into a single 2d corrected image.

One technique I tried was taking the one that had the minimum distance (and was therefore closest to good data).

The Matlab program `xymcor` does this, Figure A-5(b) shows it.

6.3.5 Weighted Average

Another thing you can do with the distance maps is to use them as an inverse weighting to a weighted average of x and y interpolated data. If the distance maps are $d_x[i,j]$ and $d_y[i,j]$, and the corrected data is $c_x[i,j]$ and $c_y[i,j]$, then the weighted average of the data $c_{2d}[i,j]$ is:

$$c_{2d}[i,j] = \frac{c_x[i,j]d_y[i,j] + c_y[i,j]d_x[i,j]}{d_y[i,j] + d_x[i,j]}$$

The Matlab program `xywcor` does this interpolation. It's not too dissimilar to stretching a membrane across the good data. Figure A-5(c) shows it.

6.4 Experimental Results

I tried all these algorithms on the images we took, shown in Figure A-1.

The thresholded depth maps in Figure A-3 show the result of using the different error metrics and keeping the best half of the data. It's hard to say which error metric was the best. The parabolic seemed to strike a good balance between emphasizing the strong data around edges and keeping low level texture data. None of them could tackle the single side peak problem.

The error correcting algorithms are shown in Figures A-4 and A-5. They were applied using the parabolic curvature error metric. Only doing one dimensional correction gives you a very streaky looking picture. The two dimensional algorithms fill it in better. There are still problems visible caused by bad data being interpreted as good data. Of all the problems, the single side peak is the most tenacious one, and shows up next to all the text, around the bar code, and on the doorway behind the bottle.

To compare error correction with a small window to using a big window with no correction, I processed the scene using a big 21x21 contrast summation window. The result is in Figure A-6. You can see that the depth map is beautifully homogenous, at least around the boxes, and the reconstructed image looks quite noise free. The only problem with the reconstructed image is the slight blurriness around the game edges as they meet the table. This is caused by blurring of the depth data in that region since the window is so large. Even at 21x21, the window isn't quite big enough to bridge all the empty spaces. Applying a simple parabolic error metric and the minimum distance xy error correction in Figure A-6(b) makes it look really gorgeous.

Perhaps a system needs to be developed that uses an adaptive window for contrast summation, bigger when there is less data and smaller when there is more. That might be able to keep the edges sharp and squeeze the faint texture data out of large blank looking areas.

Chapter 7

Estimating Time Required

Is any of this really implementable in real time? Can we get multiple depth maps per second with present technology?

We're going to estimate the number of operations per second of processing power we would need to implement a system with linear scaling on the incoming images, a parabolic curve fit, a magnitude error metric, and minimum distance xy error correction.

We'll get an answer expressed in these parameters:

G How many images you take per processing which is how many points you want in the contrast curve.

F The number of output frames per second you want.

w The width of each image.

h The height of each image.

We can establish lows and highs for each parameter. A system whose parameters were all low would be poor. A system whose parameters were all high would be amazing.

Parameter	low	high
G	7	20
F	8	30
w	256	512
h	256	512

For added flexibility we'll first make the time estimates in terms of type of operation, rather than grouping all operations together or trying to assign cycle times to each kind of operation.

We're not going to count access operations, and assume that they can be implicitly folded into the arithmetic operators, since those have to access their operands anyway. These are the kinds of operations we're going to be concerned with:

A Addition, subtraction and comparison, probably as fast as accesses.

M Multiplication, conceivably as fast as additions and accesses.

D Division, slower than multiplies.

The times computed here are approximate since much is implementation and architecture specific, but should provide us a pretty good feel for it. Also, many of these operations can be implemented in hardware as part of a pipeline with a throughput as fast as the incoming data. In fact they were specifically designed to be so. But assuming we had to do it all in software, here's what we'd get.

7.0.1 Time for Linear scaling

Assume that each incoming frame must be scaled by some constant close to 1. Linearly interpolating every pixel requires two multiplies and two subtractions.

$$\begin{aligned} A &= 2GFwh \\ M &= 2GFwh \end{aligned}$$

This table evaluates these times for "low" and "high" systems.

	A	M
low	7.3 mops	7.3 mops
high	314.6 mops	314.6 mops

7.0.2 Time for Parabolic Curve fitting

Given a curve you find the maximum and then fit it and the surrounding two points to the parabola, which as we said before takes 6 multiplies, 7 additions, and one division, as well as G comparisons.

$$\begin{aligned} A &= (G+7)Fwh \\ M &= 6Fwh \\ D &= wh \end{aligned}$$

Which gives us:

	A	M	D
low	7.34 mops	3.14 mops	.524 mops
high	212 mops	47.1 mops	7.86 mops

7.0.3 Time for Magnitude Error Metric

We'll assume that we have it already since we needed the magnitude to get the parabolic curve fit. This is alot faster and just about as effective as the parabolic curvature metric.

7.0.4 Time for Minimum Distance X and Y Error Correction

We'll assume that our data is decent and only bridge 30 percent of it is bad. Each pixel bridged over takes 2 additions, and each cluster takes a subtraction and a division. Let's say the average cluster size is 5. Each pixel needs to have its distances compared, and have its error metric compared to see if it needs to be bridged.

$$\begin{aligned} A &= (2+1+.3/5+(2\cdot .3))wh \\ D &= (.3/5)wh \end{aligned}$$

	A	D
low	.24mops	.004 mops
high	.96mops	.015 mops

7.0.5 Summary of Times and Example

We see the totals here:

	A	M	D
low	14.89 mops	10.44 mops	.528 mops
high	527 mops	361.7 mops	8.01 mops

Now we're going to add up the times for the different operations to arrive at a single numbers for the "low" and "high" systems. We'll say that multiplies are as fast as additions, and divisions are a fifth as fast.

Doing that math out, we get a low system speed of 28 mops, and a high system speed of 929 mops. Counting overhead, I'd multiply all speeds by about four to get what you'd actually need in terms of Hz. So you'd need about 112MHz to power the low end system and 3.7GHz for the high end system. Neither is inconceivable, especially with parallel processing and custom hardware.

Notice that about half the time is spent scaling. A custom hardware scaler, optical scaler, or no scaling could reduce the needed speed significantly. This is a compleling argument for using the Alvarez lens. Also, it's interesting to note that its cheaper to increase G, the number of points in your contrast curve, than it is to increase either dimension or frame per second count. The higher G is, the more accurate the depth measurement. So even if you don't have a system with high spatial resolution in x and y, you can probably get a pretty accurate depth measurement.

Chapter 8

Conclusion

We've explored fast ways of fitting parametric curves to the measured contrast curves, and found that the gaussian and straight line fits reduce stair-stepping more than the usual parabolic fit, at least when your images are taken far enough apart in *sl* space such that r_c gets to be about 1 between images. Ideally, you should choose the kind of curve to use as a fit depending on just how much r_c is allowed to grow between frames. This insight was largely afforded to us by developing the model of the contrast curve.

Mapping the focus parameter into a space linear in r_c did not seem necessary, particularly when peak finding is performed on a small number of points close together.

We tried several error metrics and found most of them unable to find the really hard case, the case where the single side peak is alone on the curve and gets mistaken for the central peak. Development of the model allowed us to see why these side peaks occur. This is the one big case where using big contrast summation windows is has clear superiority over the error correction techniques. However it is hoped that with better curve recognition, the side peaks can be identified, and good depth information extracted from them.

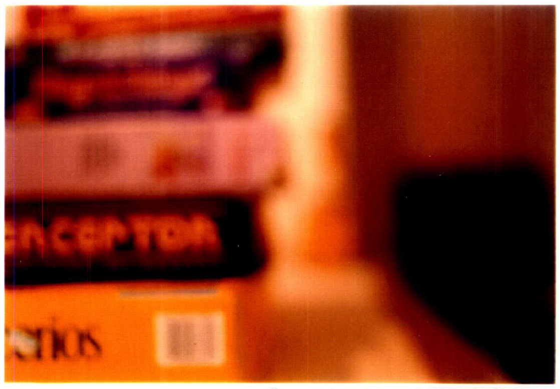
The big contrast summation window provides a less noisy depth map than we could get with a 3x3 window. However, even it benefited from error correction.

The speed analysis seemed to show that techniques presented herein could achieve several frames a second implemented with simple custom hardware or a parallel processing computer.

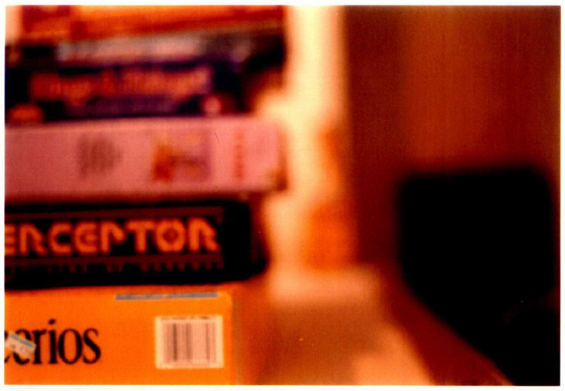
Further avenues to explore are better curve recognition and the use of adaptively sizing summation windows.

Appendix A

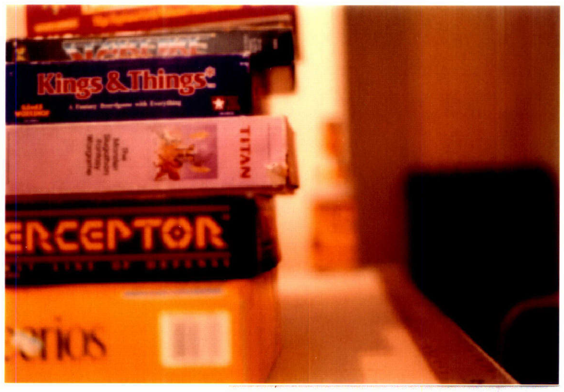
Color Plates



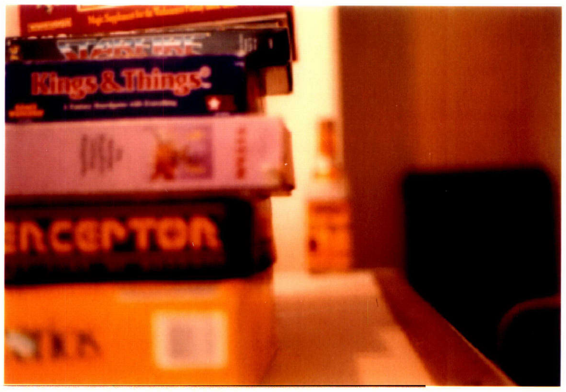
(a)



(b)

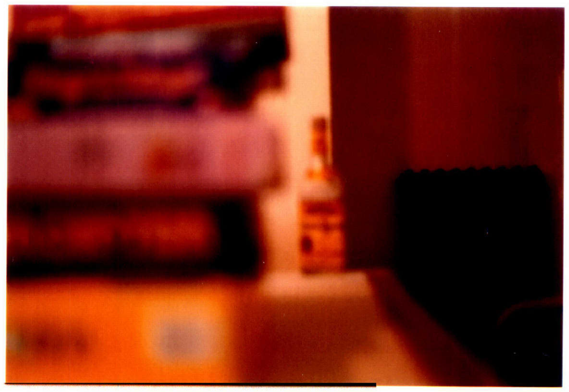


(c)



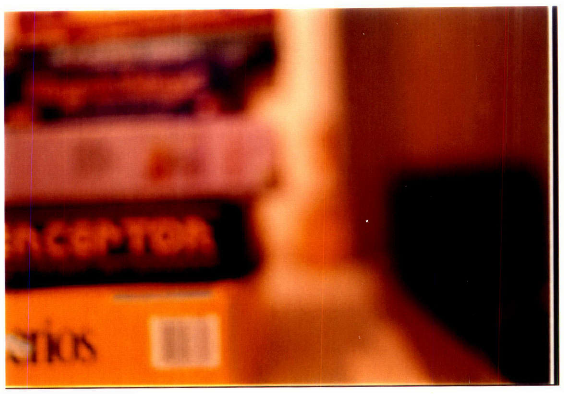
(d)

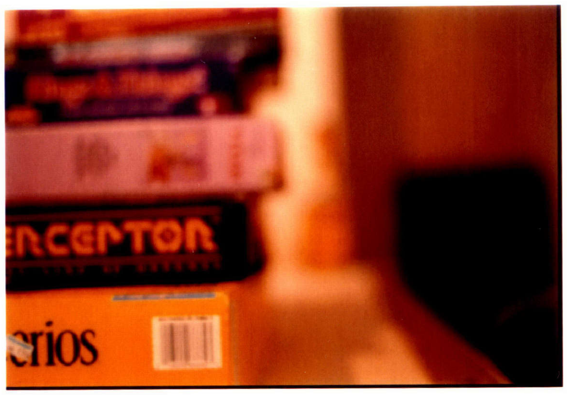
(e)

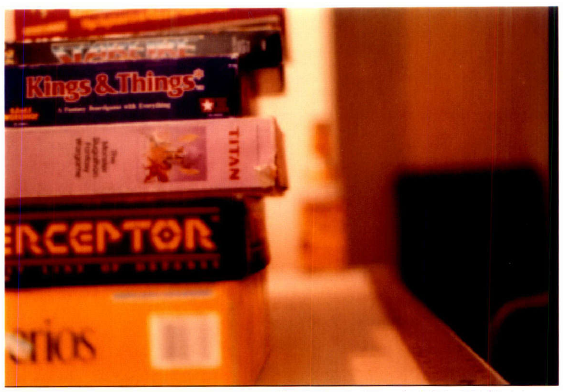


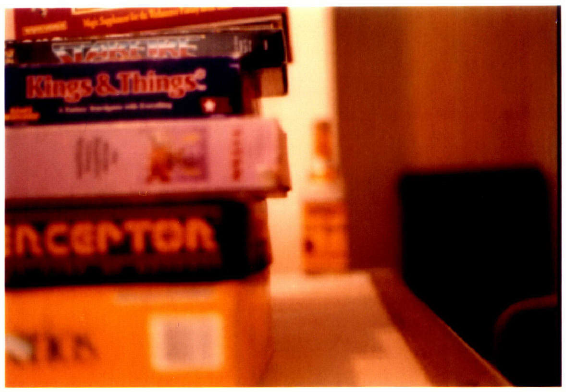
(f)

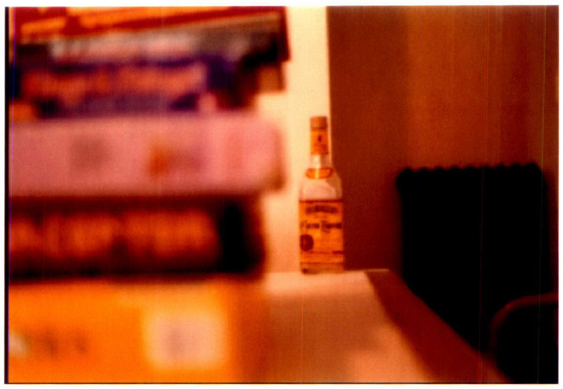
Figure A-1: Six of the source images.


(a)


(b)


(c)


(d)


(e)

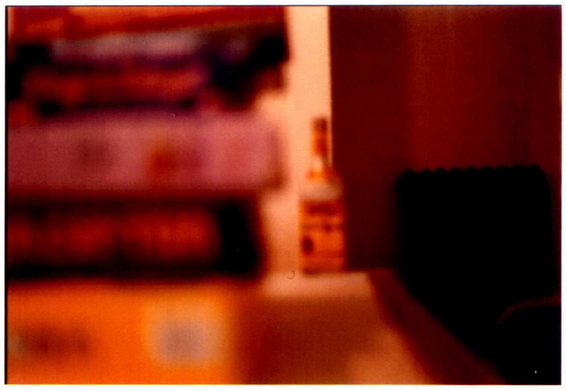
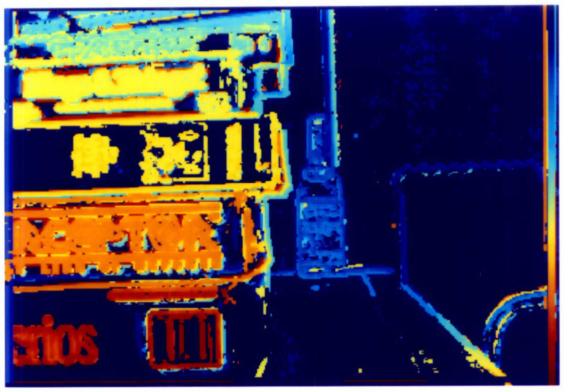
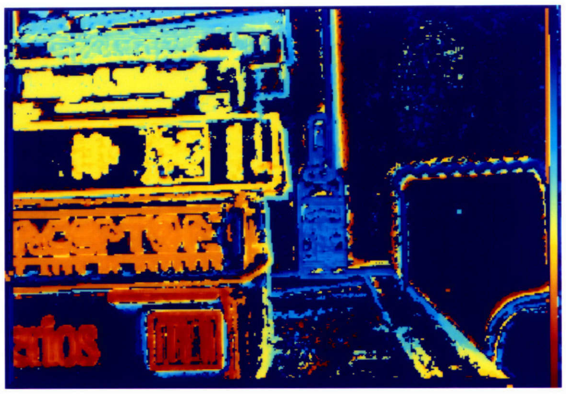

(f)

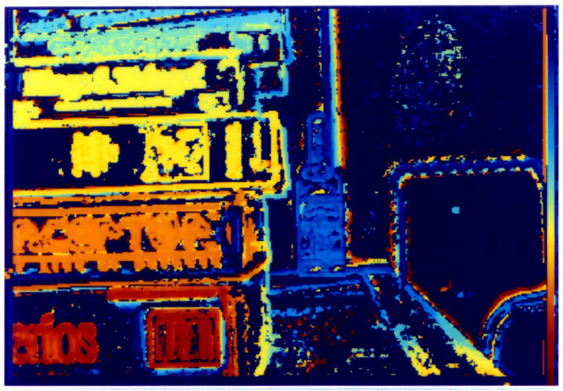
Figure A-2: Source images after fixed film scale compensation.



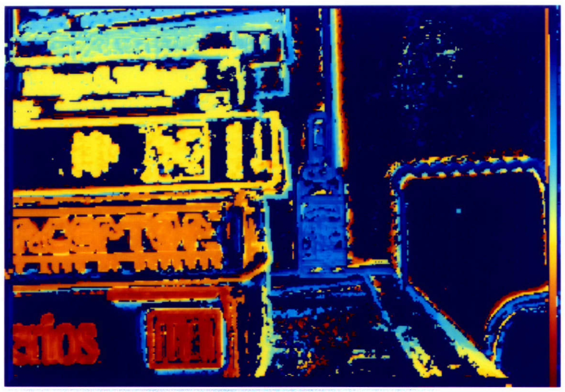
(a) Curvature, no twin peak detector.



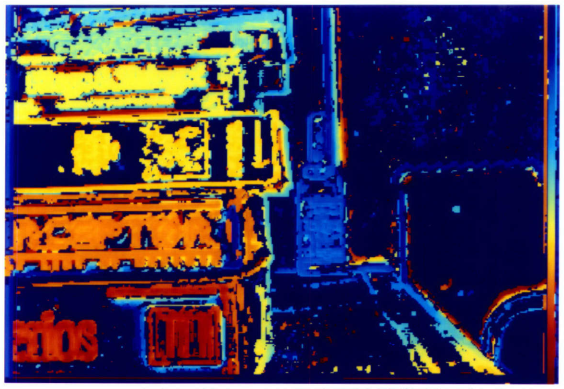
(b) Parabolic curvature.



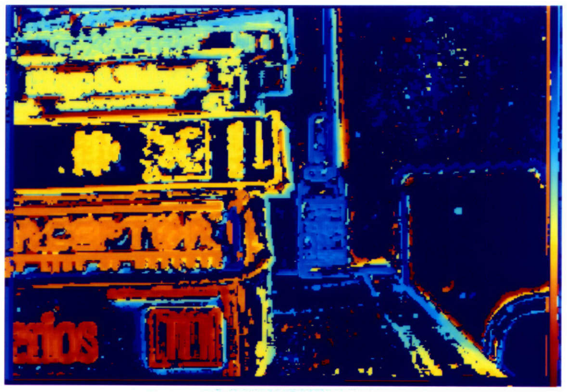
(c) Normalized parabolic curvature.



(d) Peak magnitude

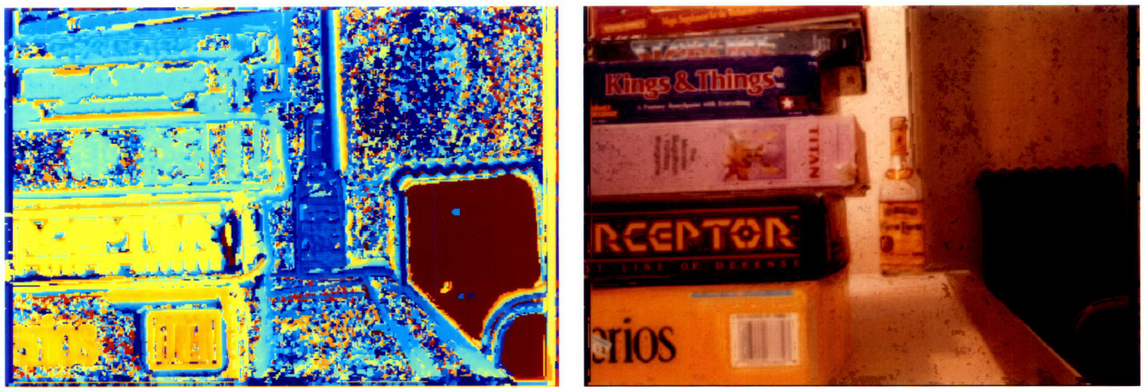


(e) Curve fit, no twin peak detector

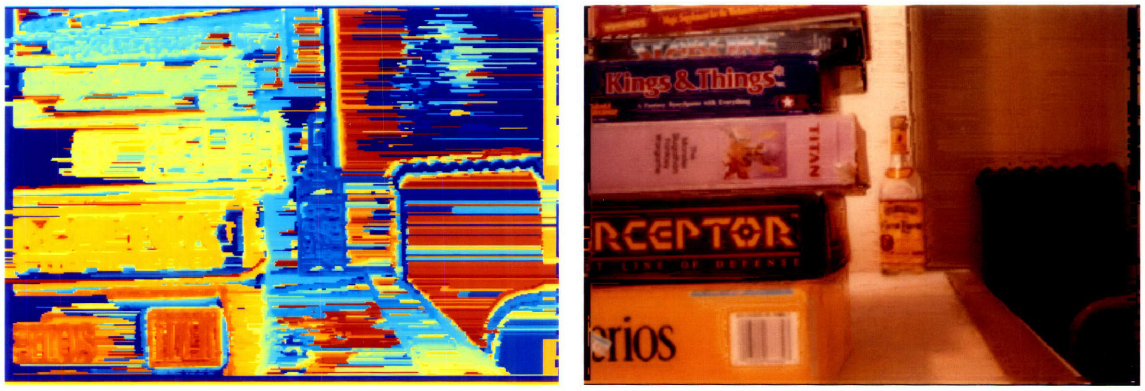


(f) Curve fit

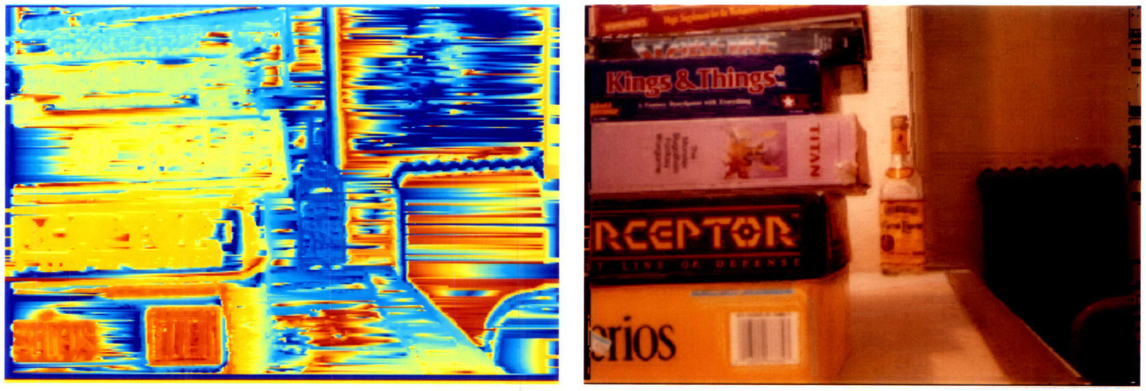
Figure A-3: Thresholded depth maps using different error metrics.



(a) Uncorrected.

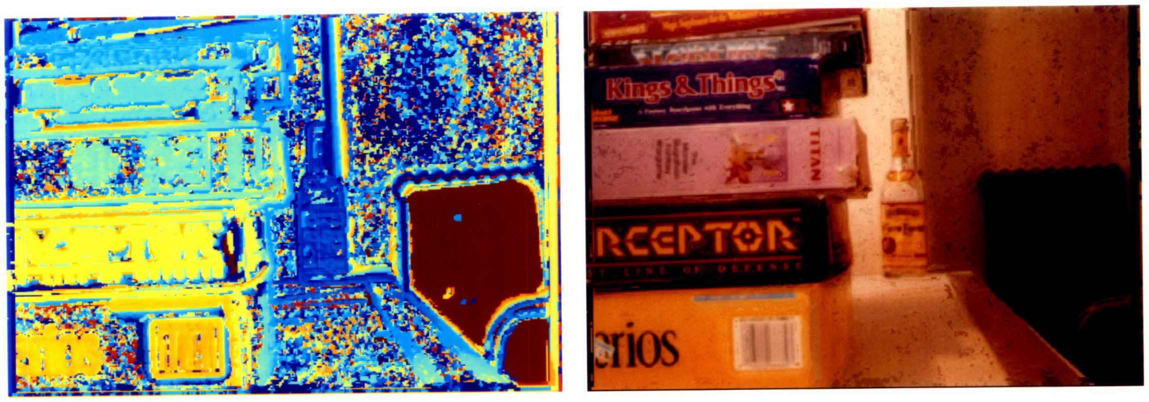


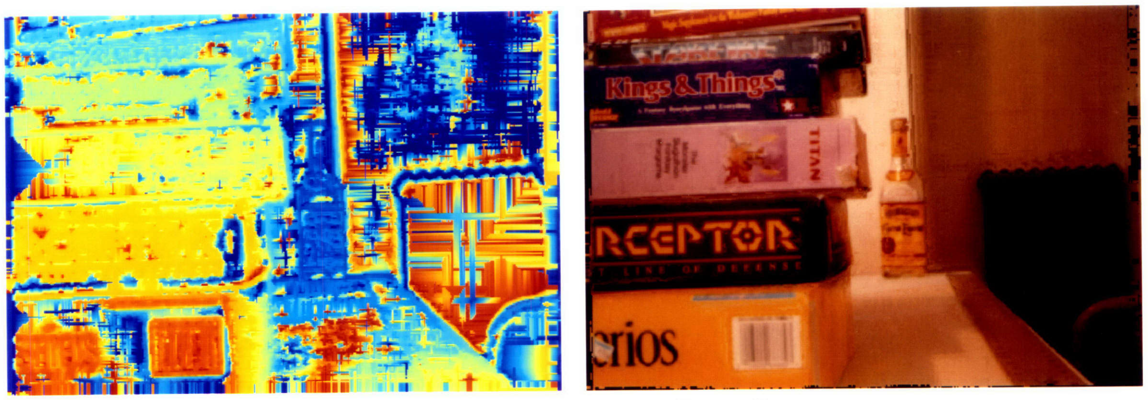
(b) Zeroth order hold in x.



(c) Linear interpolation in x.

Figure A-4: One dimensional error correction applied to depth map and collage. Used parabolic curvature error metric.


(a) Uncorrected.


(b) Minimum distance of x and y.

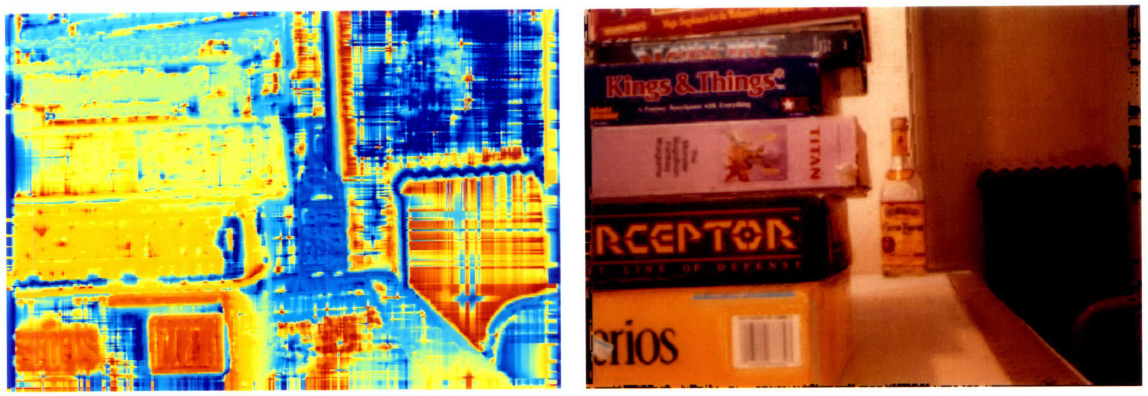
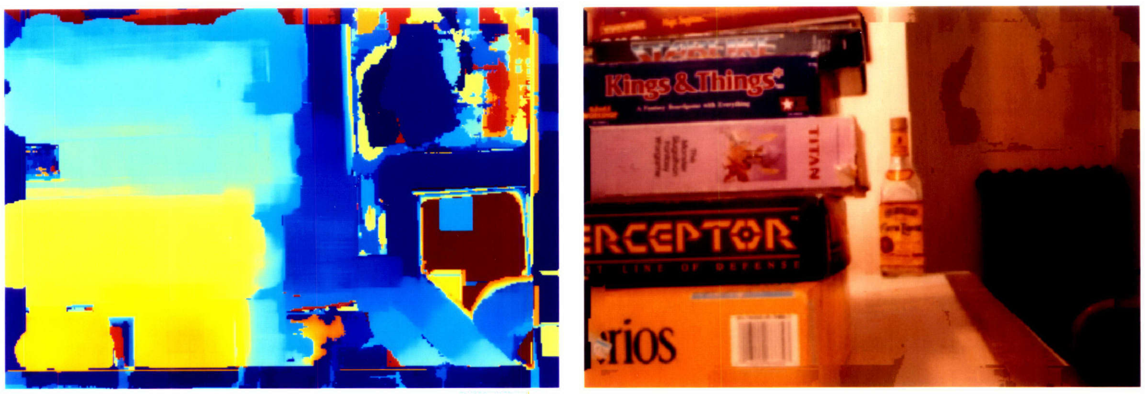
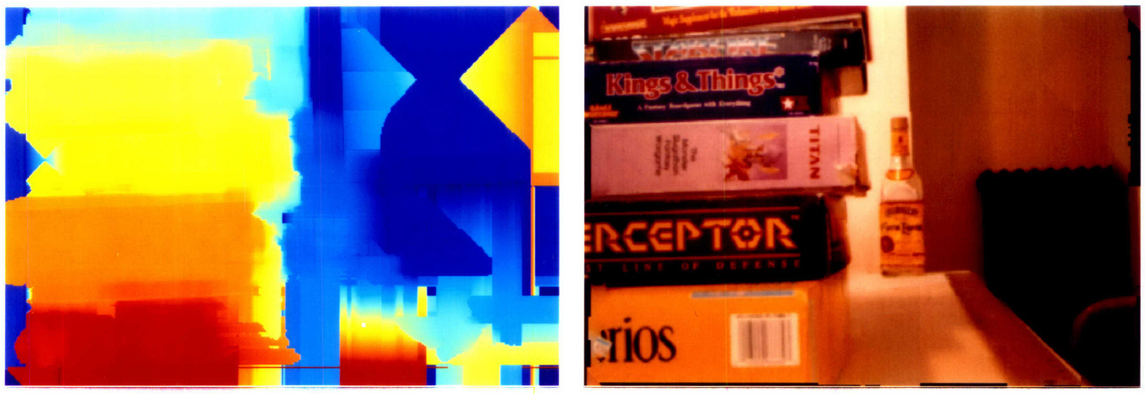

(c) Weighted distance of x and y.

Figure A-5: Two dimensional error correction applied to depth map and collage. Using parabolic curvature error metric.



(a) Uncorrected.



(b) Corrected using curvature metric and minimum distance of x and y.

Figure A-6: Depth and collage images generated with a 21x21 contrast summation window. Note that even with a large window it benefits from error correction.

Appendix B

Matlab Code

```
function x = flensmag(sl,f)
% function x = flensmag(sl,f)
% takes a vector of lens distances and lens focal length,
% returns a vector x of scaling factors where > 1 means
% to magnify the images and < 1 means to shrink them
% scales them to sl(1).
% For a fixed lens system.

%divide each element of sl by sl(1).  Not exactly hard.

x = sl / sl(1);
```

```
function x = mlensmag(sl,f)
% function x = mlensmag(sl,f)
% takes a vector of lens distances and lens focal length,
% returns a vector x of scaling factors where > 1 means
% to magnify the images and < 1 means to shrink them
% scales them to sl(1), which is not specified, being 1

% get average sl of pairs
l = size(sl,2);
slav = (sl(1:l−1) + sl(2:l))/2;
% common distances
dav = (slav.*slav)./(slav − f);

% assign pairs then get the products
```

```
ps = (sl(1:l−1).*(dav − sl(2:l)))./(sl(2:l).*(dav − sl(1:l−1)));

% cumprod it
x = cumprod(ps);
```

```
function [xp,yp,a] = parpeak(x,y)
% function [xp,yp,a] = parpeak(x,y)
% returns the peak of a vector of x y pairs
% using parabolic interpolation on the max
% and two neighboring ones
% returns x location of peak in xp and curviness in a

[v,i]  = max(y);

[a,p] = isbad(y);
if (a)
  xp = x(p);
  yp = 0;
  a = 0;
  return;
end;

s = i−1:i+1;
v = x(s);

c = [ (v.*v)'  v'  ones(3,1) ] \ y(s)';

a = −c(1);
xp = c(2)/(2*a);
yp = c(3) − a*xp*xp;
```

```
function [xp,yp,a] = parpeak(x,y)
% function [xp,yp,a] = parpeak(x,y)
% returns the peak of a vector of x y pairs
% using parabolic interpolation on the max
% and two neighboring ones
% returns x location of peak in xp and curviness in a

[yp,i]  = max(y);

[a,p] = isbad(y);
if (a)
   xp = x(p);
   yp = 0;
   a = 0;
   return;
end;

a = 0;
for p1=i+1:max(size(y))
  if (y(p1)<.25*yp) a=1; break; end;
end;

for p2=i−1:−1:1
  if (y(p2)<.25*yp) a=a+1; break; end;
end;

if (a<2)
  a = 0;
  xp = x(i);
  return;
end;

a = yp;
xp = (y(p2)*x(p2) − y(p1)*x(p1) − abs(x(p1)−x(p2))*sqrt(y(p1)*y(p2)))/(y(p2)−y(p1));
```

```
function [xp,yp,a] = parpeak(x,y)
% function [xp,yp,a] = parpeak(x,y)
% returns the peak of a vector of x y pairs
% using parabolic interpolation on the max
% and two neighboring ones
% returns x location of peak in xp and curviness in a

[yp,i] = max(y);

[a,p] = isbad(y);
if (a)
  xp = x(p);
  yp = 0;
  a = 0;
  return;
end;

a = 0;
for p1=i+1:max(size(y))
  if (y(p1)<.25*yp) a=1; break; end;
end;

for p2=i-1:-1:1
  if (y(p2)<.25*yp) a=a+1; break; end;
end;

if (a<2)
  a = 0;
  xp = x(i);
  return;
end;

a = yp;
xp = (y(p1)*x(p1) + y(p2)*x(p2))/(y(p1)+y(p2));
```

```
function [xp,yp,a] = slpeak(x,y)
% function [xp,yp,a] = slpeak(x,y)
% returns the peak of a vector of x y pairs
% using a straight line interpolation between the
% max point and two neighboring ones
% returns x location of peak in xp and y in yp along with slope measure in a

% turn on for debugging
duh = 0;

[v,i]  = max(y);

% check for badness
[a,p] = isbad(y);
if (a)
  xp = x(p);
  yp = 0;
  a = 0;
  return;
end;



s = i−1:i+1;
% says 2 to 1 is less steep than 2 to 3
if ( abs((y(i)−y(i−1))/(x(i)−x(i−1))) < abs((y(i)−y(i+1))/(x(i)−x(i+1))) ) s = fliplr(s); end;

u = x(s);
v = y(s);

a = (v(1) − v(2)) / (u(1) − u(2));
b1 = (u(1)*v(2) − u(2)*v(1)) / (u(1)−u(2));
b2 = v(3) + a*u(3);

xp = (b2−b1)/(2*a);
yp = (b2+b1)/2;
a = abs(a);

if (duh==1)
  if (u(1) > u(3)) u = fliplr(u); end;
  t = u(1):.1:u(3);
  clf
  hold off
  plot(x,y,'o');
  hold on
  plot(xp,yp,'x');
  fprintf(1,'xp = %g\n',xp);
  v = yp - a*abs(t-xp);
  plot(t,v);
  xlabel('sl');
  ylabel('blur measure');
  title(sprintf('Straight line peak finder = %g',p));
  drawnow;
  if (yp < max(y)) pause; end;
end
```

```
function a = x0cor(g,h,t)
%function a = x0cor(g,h,t)
% g: depth map
% h: prob map
% t: percentage threshhold
% performs error correction
% on g using 0th order x interp
% returning result in a

t = fthresh(h,t,.001);

a = zeros(size(g));
m = max(max(g));

for i = 1:size(g,2)
  a(:,i) = x0corl(g(:,i),h(:,i),t,m);
  if (rem(i,10)==1) i; end;
end;
```

```
function a = x0corl(g,h,t,m)
%function a = x0corl(g,h,t,m)
% g: depth map line
% h: prob map line
% t: h threshhold
% m: default value if no data
% performs error correction
% on g using 0th order x interp
% returning result in a

a = zeros(size(g));
h = (h>t);

%find first non zero h for starting
i = 1;
l = m;
while( (i<size(h,1)) & ~h(i) ) i = i+1; l=g(i); end;

for i = 1:size(g,1)
  if (h(i))
    a(i) = g(i);
    l=g(i);
  else
    a(i) = l;
  end;
end;
```

```
function [a,d] = x1cor(g,h,t)
%function [a,d] = x1cor(g,h,t)
% g: depth map
% h: prob map
% t: percentage threshhold
% performs error correction
% on g using a linear interp
% returning result in a, and
% distance map in d

t = fthresh(h,t,.001);
```

```
a = zeros(size(g));
d = zeros(size(g));
m = max(max(g));

for i = 1:size(g,2)
  [a(:,i),d(:,i)] = x1corl(g(:,i),h(:,i),t,m);
  if (rem(i,10)==1) i; end;
end;
```

```
function [a,d] = x1corl(g,h,t,m)
%function [a,d] = x1corl(g,h,t,m)
% g: depth map line
% h: prob map line
% t: h threshhold
% m: default value if no data
% performs error correction
% on g using 0th order x interp
% returning result in a, and
% distance from good data in d

a = zeros(size(g));
d = zeros(size(g));
h = (h>t);

%find first non zero h for starting
i = 1;
l = m;
li = 0;
while( (i<size(h,1)) & ~h(i) ) i = i+1; l=g(i); end;

for i = 1:size(g,1)
  if (h(i))
    a(i) = g(i);
    % fill in missing area
```

```
      if (i–li > 1)
        a(li+1:i–1) = l + (1:i–li–1)*(g(i)–l)/(i–li);
        dl = floor((i–li)/2);
        d(li+1:li+dl) = 1:dl;
        d(i–dl:i–1) = dl:–1:1;
      end;
      l = g(i);
      li = i;
    end;
  end;

  if (li < i)
    a(li+1:i) = l*ones(i–li,1);
    d(li+1:i) = 1:i–li;
  end;
```

```
function [a,d] = xymcor(g,h,t)
%function [a,d] = xymcor(g,h,t)
% g: depth map
% h: prob map
% t: percentage threshhold
% performs error correction
% on g using a minimum dist of x and y
% returning result in a, and
% min distance map in d

t = fthresh(h,t,.001);

a = zeros(size(g));
d = zeros(size(g));
a1 = zeros(size(g));
d1 = zeros(size(g));
m = max(max(g));
```

```
% get x matrix
for i = 1:size(g,2)
  [a(:,i),d(:,i)] = x1corl(g(:,i),h(:,i),t,m);
  if (rem(i,10)==1) i; end;
end;

% get y matrix
for i = 1:size(g,1)
  [duh1,duh2] = x1corl(g(i,:)',h(i,:)',t,m);
  a1(i,:) = duh1';
  d1(i,:) = duh2';
  if (rem(i,10)==1) i; end;
end;

% now combine them, yes!
for i = 1:size(g,1)
  for j = 1:size(g,2)
    [m,mi] = min([d(i,j) d1(i,j)]);
    if (mi==2)
      d(i,j) = m;
      a(i,j) = a1(i,j);
    end;
  end;
end;
```

```
function [a,d] = xywcor(g,h,t)
%function [a,d] = xywcor(g,h,t)
% g: depth map
% h: prob map
% t: percentage threshhold
% performs error correction
% on g using a weighted inverse
% dist of x and y
% returning result in a, and
% min distance map in d

t = fthresh(h,t,.001);

a = zeros(size(g));
d = zeros(size(g));
a1 = zeros(size(g));
d1 = zeros(size(g));
m = max(max(g));

% get x matrix
for i = 1:size(g,2)
  [a(:,i),d(:,i)] = x1corl(g(:,i),h(:,i),t,m);
  if (rem(i,10)==1) i; end;
end;

% get y matrix
for i = 1:size(g,1)
  [duh1,duh2] = x1corl(g(i,:)',h(i,:)',t,m);
  a1(i,:) = duh1';
  d1(i,:) = duh2';
  if (rem(i,10)==1) i; end;
end;

% now combine them, yes!
for i = 1:size(g,1)
  for j = 1:size(g,2)
    [m,mi] = min([d(i,j) d1(i,j)]);
    if (m==0)
      if (mi==2)
        d(i,j) = m;
        a(i,j) = a1(i,j);
      end;
    else
      s = 1/d(i,j) + 1/d1(i,j);
      a(i,j) = (a(i,j)/d(i,j) + a1(i,j)/d1(i,j))/s;
      % weighted average of two distances?
      d(i,j) = 2/s;
    end;
  end;
end;
```

```
function [xp,yp,a] = fitpeak(x,y)
% function [xp,yp,a] = fitpeak(x,y)
% finds the center and peak using
% two straight lines then returns a
% based on how well it fits 1/r^2 curve.
% pretty scary.


[v,i] = max(y);

[a,p] = isbad(y);
if (a)
  yp = 0;
  xp = x(p);
  a = 0;
  return;
end;

s = i-1:i+1;
if (abs((y(i)-y(i-1))/(x(i)-x(i-1))) < abs((y(i)-y(i+1))/(x(i)-x(i+1))) ) s = fliplr(s); end;

u = x(s);
v = y(s);

a = (v(1) - v(2)) / (u(1) - u(2));
b1 = (u(1)*v(2)-u(2)*v(1)) / (u(1)-u(2));
b2 = v(3) + a*u(3);
```

```
xp = (b2-b1)/(2*a);
yp = (b1+b2)/2;

% scale y to 1
y = y/yp;

% approximate mapping from sl to rc
x = 2.1*(x - xp);

% take only points in 1/r^2 region
mask = (abs(x)>1);

y1 = 1./(4*x.*x);
% take distance squared
d = ((y1 - y) .* mask) .^ 2;

% take area under it, don't really need .5 scaling
w = size(d,2);
a = sum(.5*(d(1:w-1) + d(2:w)).*(x(1:w-1) - x(2:w)));

%clf
%hold off
%plot(x,y,'o');
%hold on
%plot(x,y1.*mask,'x');d(2:w)).*(x(1:w-1) - x(2:w)));
```

Bibliography

[1] Howard Stuart, The Optical Sensing of Three-Dimensional Surfaces, Master's Thesis, *Massachusetts Institute of Technology*, 1990.

[2] John Delisle, Three Dimensional Profiling Using Depth of Focus, Master's Thesis, *Massachusetts Institute of Technology*, 1991.

[3] Stephen Scherock, Depth from Defocus of Structured Light, Master's Thesis, *Massachusetts Institute of Technology*, 1991.

[4] Paul Fieguth, Range Estimation Accuracy Using Depth-From-Focus Methods – Theory and Experiment, *Massachusetts Institute of Technology*, 1992.

[5] F.T. Arecchi, D. Bertani, and S. Ciliberto. A fast versatile optical profilometer. *Optics Communications*, 31(3):263-266, December 1979.

[6] D.Bertani, M. Cetica, and S. Ciliberto. A fast optical profilometer. *Optics Communications*, 46(1):1-3, June 1983.

[7] J. Mignot and C. Gorecki. Measurements of surface roughness: Comparison between a defect-of-focus optical technique and the classical stylus technique. *Wear*, 87:39-49, 1983.

[8] K. Engelhardt and G. Hausler. Acquisition of 3-d data by focus sensing. *Applied Optics*, 27(22):4684-4689, November 1988.

[9] J. Kagami, T. Hatazawa, and K. Koike. Measurement of surface profiles by the focusing method. *Wear*, 134:221-229, 1989.

[10] Reg G. Willson and Steven A. Shafer. What is the Center of the Image? Technical Report CMU-CS-93-122, School of Computer Science, Carnegie Mellon University, 1993.

[11] Reg G. Willson and Steven A. Shafer. Dynamic Lens Compensation for Active Color Imaging and Constant Magnification Focusing. Technical Report CMU-RI-TR-91-26, Robotics Institute, Carnegie Mellon Institute, 1991.

[12] Yalin Xiong and Steven A. Shafer. Depth from Focusing and Defocusing. Technical Report CMU-RI-TR-93-07, Robotics Institute, Carnegie Mellon Institute, 1993.

[13] M. Subbarao, T. Choi, SUNY/Stony Brook; A. Nikzad, Symbol Technologies Inc., "Focusing techniques," *Machine Vision Applications, Architectures, and System Integration,* Bruce G. Batchelor, Susan Snell Solomon, Frederick M. Waltz, Editors, Proc. SPIE 1823, 163-174 (1992)

[14] M. Subbarao, Ming-Chin Lu, SUNY/Stony Brook, "Computer modelling and simulation of camera defocus," in *Optics, Illumination, and Image Sensing for Machine Vision VII,* Donald J. Svetkoff, Editor, Proc. SPIE 1822, 110-120 (1993)

[15] M. Subbarao, T. Choi, SUNY/Stony Brook, "New method for shape from focus," in *Machine Vision Applications, Architectures, and Systems Integration II,* Bruce G. Batchelor, Susan Snell Solomon, Frederick M. Waltz, Editors, Proc. SPIE 2064, 74-85 (1993)

[16] Shree K. Nayar. Shape from Focus. Technical Report CMU-RI-TR-89-27, Robotics Institute, Carnegie Mellon Institute, 1989.

[17] Eugene Hecht, *Optics*, Addison-Wesley, Reading, Mass., 1990.