Height and Gradient from Stereo Shaded Images

by

Kenneth A. Walker

Submitted to the Department of Electrical Engineering and Computer Science

in Partial Fulfillment of the Requirements for the Degrees of

Bachelor of Science in Computer Science and Engineering

and Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

May 24, 2002

[June 2002]

Author____

Department of Electrical Engineering and Computer Science
May 24, 2002

Certified by_____________________________

Berthold K. P. Horn
~~Thesis Supervisor~~

Accepted by__________

Arthur C. Smith
Chairman, Department Committee on Graduate Theses

Height and Gradient from Stereo Shaded Images
by
Kenneth A. Walker

Submitted to the
Department of Electrical Engineering and Computer Science

May 24, 2002

In Partial Fulfillment of the Requirements for the Degree of
Bachelor of Science in Computer Science and Engineering
and Master of Engineering in Electrical Engineering and Computer Science

Abstract

Stereo photogrammetric algorithms lack the ability to resolve local surface texture as a result of the need for smoothness of depth. Shape-from-shading methods give better estimates of fine surface detail, but they suffer from problems with non-unique solutions, especially when the boundary conditions are not known. The combined iterative algorithm presented provides a more accurate and stable method for recovering a digital elevation model from stereo aerial image pairs using shape-from-shading information and stereo geometry. A monocular shape-from-shading algorithm is extended to make use of stereo matching, using an aggressive warping process to adjust the height-and-gradient solution at features where there is a relatively high stereo error between the two images. This allows the accurate recovery of both local and global characteristics of the terrain. The algorithm is shown working on synthetic epipolar-plane images of constant albedo with a known reflectance function.

Thesis Supervisor: Berthold K. P. Horn
Title: Professor of Electrical Engineering and Computer Science

Contents

List of Figures

1 Introduction

The ability to generate accurate digital elevation models (DEM) of a given land-area from remote imagery is essential when on-site surveying is not practical. This problem is especially applicable to the mapping of distant planets where all data comes from orbiting satellite probes. For exploration purposes, it is just as important to detect small surface features as it is to correctly determine the elevation of large ones. In addition to locating ridges, valleys, impact craters and other major features, mapping small changes in the surface gradient can tell a spacecraft if a local area is flat enough for a landing or tell a rover if it the terrain is too rough to pass through. With that in mind, the goal of this project is to develop a method for remote photogrammetry that gives the highest possible accuracy detecting features of all sizes.

The best way to correctly determine the elevation of large features is by using geometric methods such as binocular stereo. By knowing the camera geometry, and correctly matching common regions in each photo, the absolute position of those regions can be found. The matching problem is made more difficult when there are a number of suitable matches or no exact matches for regions in the two images. As a result, binocular stereo is less useful in areas without sufficient detail to reliably find local region matches. Furthermore, there must be some smoothness assumption in order for the matching process to work, and this limits the amount of detail that can be resolved with this approach.

Complementarily, the shape-from-shading approach combines the brightness of image pixels with known reflectance properties of the surface to estimate the surface gradient. Shape from shading works well for smooth surfaces or small details, though the method often has problems due to multiple solutions, or relative inaccuracies accumulated over distance. At best, shape from shading can give a detailed, relative elevation map. By effectively combining the benefits of Shape-from-Shading with binocular stereo, a detailed absolute elevation map could be recovered.

2 Background

2.1 Geometric Stereo

Photogrammetry from stereo is based on small differences between two images of the same subject, which are taken from slightly different positions. These differences give clues to the relative depth of points in the scene. Surfaces closer to the camera undergo greater lateral changes compared to surfaces that are farther away. These differences however, are proportional to the relative distances of both height extremes to the camera. This means that the farther the camera, the less difference, and orthographic photos taken from an "infinite" distance would not yield any useful stereo data.

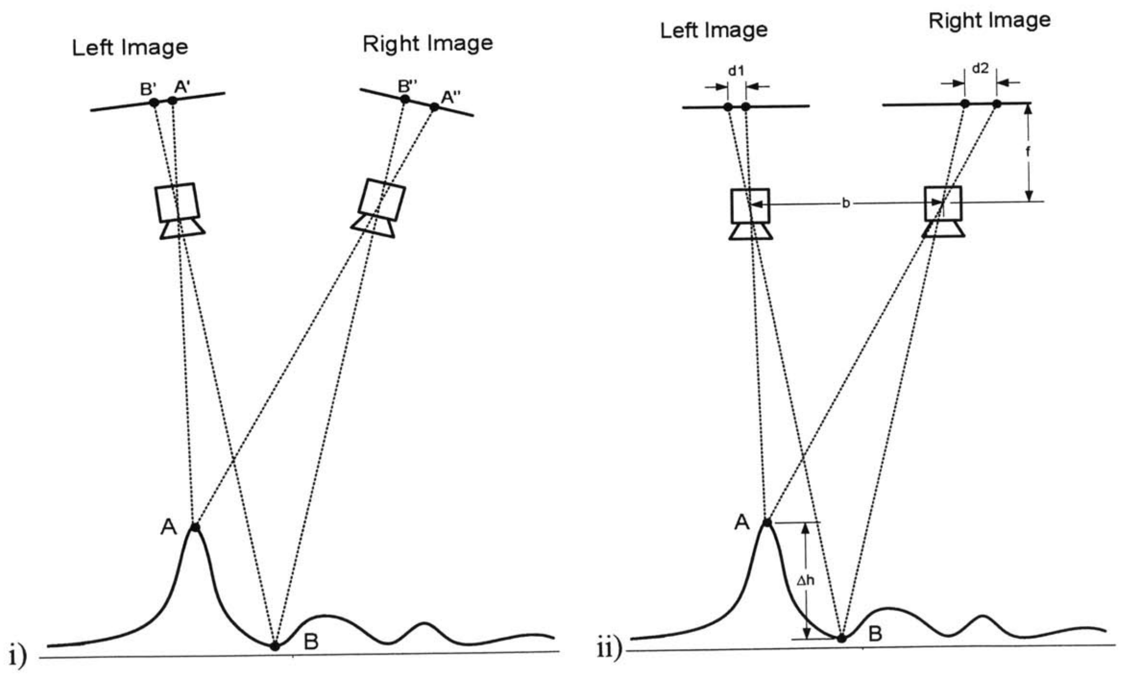


Figure 2-1: The generic and orthonormal stereo problems

Similarly, small shallow features on a surface are very difficult to detect using stereo images because the relative height difference is so tiny. Furthermore, because the stereo approach uses visual cues on the surface itself to match regions across the pair of images, features recognizable in this way must have some minimum size, which can be distinctly located on each image.

The most important step is to correctly find the mapping between points in the left and right images, since they won't be in exactly the same position. There are a variety of approaches to this open problem, including area-correlation [15], [17], edge-based, and gray-level matching [11]. The type of detail in the scene can greatly affect the matching process, since for example, a uniformly textured surface gives no stereo clues, and any match within that area can be considered as good as the other possibilities. If the cameras have parallel view vectors, then they create *epipolar-plane images*. That is, any disparity between the left and right images is confined to the horizontal direction, so the search space is reduced to the corresponding row in the other image. Most approaches introduce some additional constraints that reduce the number of potential matches. The most common constraint is *smoothness*, meaning it is assumed that the surface is continuous nearly everywhere, the pixel disparity changes slowly, and that points close together in the left image will also be close together in the right image. The smoothness constraint makes it possible to correlate local regions instead of only single pixels, thereby increasing the probability of a correct match. On the other hand, it also naturally limits the amount of detail that can be resolved. *Ordering* is another common constraint. This means that if A is left of B in one image, it can't be right of B in the other image. This is logical if we are always looking at the same side of the surface.

Once the matching is known, the surface features can be found relatively easily. In this case, the depth difference h between two points in the scene (Figure 2-1b) can be found by [2]

$$\Delta h = \frac{\Delta x * f}{b + \Delta x} \qquad \text{where} \quad \Delta x = d1 - d2 \qquad (2\text{-}1)$$

The stereo approach works regardless of lighting conditions, as long as there is some illumination on the surface. It also is not affected by variations in surface color (albedo) that might confuse a shape-from-shading scheme. In fact, it makes use of changes in surface appearance to match regions across the stereo pair. A perfectly uniform looking surface would be difficult to analyze using stereo, for the same reason.

2.2 Shape-from-shading

The premise of many shape-from-shading methods revolves around the reflectance map, a function that gives the brightness/reflectance of a particular surface based on the slope, or gradient components, of the surface at any point. Unfortunately the solution cannot be directly computed because the image is missing some information--specifically, the two components of surface gradient are combined into one value: brightness. Instead, one way to reconstruct the surface is to use Calculus of Variations and an iterative approach, minimizing the difference in brightness between the image and the solution-in-progress.

The fundamental image irradiance equation in the minimization approach to shape-from-shading is given by:

$$\mathrm{E}(\mathrm{x}, \mathrm{y}) = \mathrm{R}(\mathrm{p}(\mathrm{x}, \mathrm{y}), \mathrm{q}(\mathrm{x}, \mathrm{y})) \qquad (2\text{-}2)$$

Where E(x,y) indicates the brightness at a point (x,y) on the image, and R(p,q) is the brightness value expected according to the reflectance map, for a surface z with gradient values (p,q). The most common simulated reflectance map for matte (non-glossy) surfaces is the Lambertian map which gives the brightness of a surface patch as the cosine of the angle from the surface normal to the light source direction or

$$R(p,q) = \frac{1 + p_s p + q_s q}{\sqrt{1 + p^2 + q^2}\sqrt{1 + p_s^2 + q_s^2}} \tag{2-3}$$

$$\textit{(where } \begin{bmatrix} -p_s \\ -q_s \\ 1 \end{bmatrix} \textit{ is the direction to light source)}$$

Starting with the image E, and reflectance map R, the goal of the algorithm is to find the surface z, for which the image irradiance equation holds most accurately.

In other words, we wish to minimize the difference between the left and right sides of the irradiance equation over the area of the image. This goal is summarized by the minimization of the following error equation:

$$\iint (E(x,y) - R(p,q))^2 \, dx \, dy \tag{2-4}$$

One of the strengths of Shape-from-Shading is its sensitivity to high-frequency variations in the surface contours. It can effectively pick out gradient changes due to small surface features. On the other hand, it is not necessarily as accurate regarding low-frequency components of the solution. Often there can be large amounts of "drift" from one end of the surface to another, and while local features may be represented, one side could be much higher than another when in fact they should have the same elevation.

The Shape-from-Shading approach in general is not without its own limitations, including: multiple solutions, local minima, and non-convergence. It is possible that for certain initial conditions, the algorithms will not reach the optimal solution, but instead get stuck in a local minimum of the error function. Similarly, some inputs may not converge at all, yielding an unstable output.

2.3 Horn's Height and Gradient Scheme

In the coupled height and gradient scheme [9] from which this work is derived, the error equation is first introduced as:

$$\iint (E(x,y)-R(p,q))^2+\mu((z_x-p)^2+(z_y-q)^2)\,dx\,dy \tag{2-5}$$

where the additional term is a penalty for lack of integrability of the gradient. The solution consists of p, q, and z values at every pixel. All three of these variables are solved for iteratively.

The original problem is modified in two ways in order to make it better defined, and to assist the algorithm in arriving at a correct solution. First, a variable departure-from-smoothness penalty is added to the error equation. This has the effect of encouraging smoothness and continuity of z at the expense of brightness error. A coefficient in the error function below is used to vary the extent of this trade-off. Initially, the penalty for departure from smoothness should be high, because this will cause the surface to be relatively continuous, avoiding some problems with getting caught in local minima. If this term is left in however, the solution may not be able to converge to the correct one, especially if there are sharp corners or boundaries in the solution. So this term is gradually reduced until it reaches zero, having the effect of ever sharpening the solution as it approaches the final one. The new error function is shown here:

$$\iint [(E(x,y)-R(p,q))^2+\lambda(p_x^2+p_y^2+q_x^2+q_y^2)+\mu((z_x-p)^2+(z_y-q)^2)]\,dx\,dy \qquad (2\text{-}6)$$

The other modification involves the local linearization of the reflectance map. One of the main causes of incorrect solutions is when the algorithm gets stuck in a local minimum. This happens more frequently when the surface is complex and the reflectance map is not so close to linear in the gradient. Horn addresses this problem in [9] by using a local linear approximation to the reflectance map, which gives

$$R(p,q)\approx R(p_0,q_0)+(p-p_0)R_p(p_{0,}q_0)+(q-q_0)R_q(p_{0,}q_0)+\ldots. \qquad (2\text{-}7)$$

and has been found to greatly increase the performance of the algorithm.

2.4 Related Combinatorial Methods

There have been various attempts to combine multiple visual cues (algorithms), referred to generically as "shape-from-X" modules. Poggio, Gamble and Little [14] use Markov random fields to couple the outputs of various algorithms (stereo, motion, texture, color) with the discontinuities associated with each type of visual cue. Their goal was primarily to improve the identification of surface discontinuities.

Piecewise interpolation in [3] is used to integrate sparse edge-based stereo data and a connected segmentation diagram derived from a raw needle-map from shading in the right image. They assume a coincident light-source and viewing direction, and use a modified version of Pentland's [13] slant and tilt scheme for shading analysis.

Fua and Leclerc [6] use a mesh integration framework and a weighted energy function to formally combine information from stereo, shading, silhouettes, and hand-entered features. The weights are pre-selected for different image types.

Cryer, Tsai and Shah [4] implement a biologically inspired [7] model, using FFT filters to add the low frequency information from stereo and the high frequency information from shading. Their scheme takes the final output from each process and combines them into an improved depth map.

Dorrer and Zhou [5] start with a stereo-derived elevation model and an aerial image, using the DEM to "de-shade" the image, or remove changes in surface appearance that are not resultant from changes in slope, but rather the surface itself (albedo). This leads to reduced noise in the input to their shape-from-shading algorithm. The DEM is also used for the initial estimate.

3 Integrated Binocular Shape from Shading

3.1 Coordinate systems

For the purposes of this problem, two types of 3D coordinate systems are defined. The solution will be considered in *terrain-centered coordinates*, meaning that it will give the surface elevation z_s relative to a "sea level" reference plane[1] for any 2D coordinate within the reconstructed region of that plane. Two pin-hole cameras are placed some distance h above the x-axis, separated by a given baseline b. When considering image projection, it is most convenient to use *camera-centered coordinates*, where the origin is at the focal point of the camera, the x-y plane is parallel to the terrain reference plane, and the positive z-axis extends in the camera view direction, always perpendicular to the reference plane. In practice, the view vector may not always be exactly perpendicular, but it can be assumed without loss of generality that the input images have been rectified. Finally, there are 2-dimensional *image coordinates* that result from projecting the camera coordinates onto the focal plane, using the camera location as the center of projection. To avoid sign-reversal and scaling from camera to image coordinates, we use the reference plane as the focal plane.

3.2 Extending the Monocular Shape from Shading Algorithm

Several modifications must be made to the height-and-gradient algorithm in order to use multiple images for the input. In the monocular scheme, the camera can be placed at any distance from the terrain, and the calculations are simplified if it is assumed that the focal point is far enough away from the surface that the projection can be considered

[1] "Below sea-level" never means underwater in this context.

orthographic. This assumption cannot be made when using binocular stereo, because the pixel disparity between the two images is the primary source of information. Since the disparity scales inversely with respect to the camera-z, all information is lost at an infinite distance. To simplify the problem somewhat, we ignore the case where the two cameras are not pointing the same direction, even though that would provide some disparity when viewed from infinity, since it is better considered in a shape-from-rotation approach.

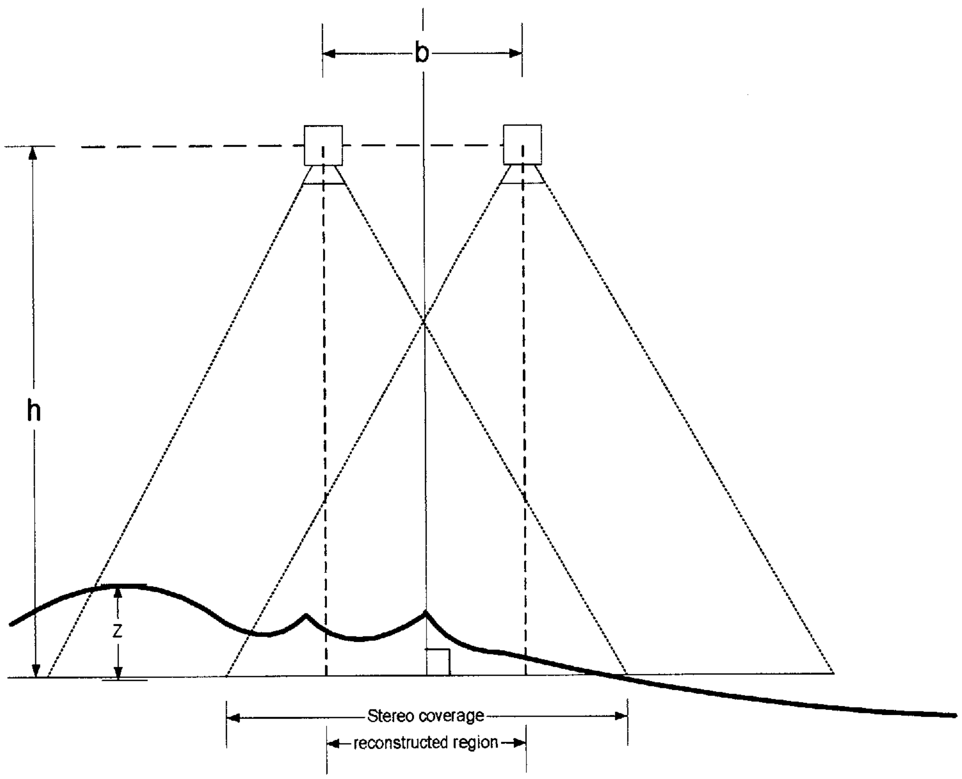


Figure 3-1: Camera Geometry

It must then be assumed that the cameras are close enough to the surface that the images are projected perspectively. Shape-from-Shading requires that we use the brightness of a terrain point, previously found from an essentially direct lookup in the image:

$$E[x, y] = \textit{image irradiance at terrain coordinates } (x,y) \tag{3-1}$$

in order to calculate the gradient at that point. With non-orthographic images, E is no longer a simple lookup. In perspective, the image coordinates of ground pixels are scaled inversely with the distance z of the ground from the camera:

$$E(x,y) = E[x_i, y_i] \qquad x_i = f\frac{x_c}{z_c} \text{ and } y_i = f\frac{y_c}{z_c} \tag{3-2}$$

where f is the focal length between the focal point and the image plane. In order to simplify the computation, $f=h$ is used here, so that one unit in the image is the same as one unit in the terrain at sea-level. Because surface elevation is not initially known at any point, it is not possible to directly calculate which image pixel corresponds to some ground coordinate, and thereby correctly find the brightness at the given ground point. In order to get started, an approximation of the surface must be used to get the brightness close to that ground point. The sea-level reference plane is chosen for the first-round approximation. Because the terrain surface is continuous and the algorithm is iterative, this approximation improves as refinement of the solution progresses.

In the orthographic case, it was sufficient to provide input images the exact dimensions of the reconstructed region, since there was a direct mapping to the terrain. When using perspective projection, the images must be slightly larger in order to guarantee that a brightness value can be found for every terrain coordinate. For example, if the terrain is above sea-level, then the pixels corresponding to the boundary would be outside an image of exactly the same dimensions. For practicality, it is assumed that the range of elevation can be known to fall within some finite bounds, and the required minimum input size can be determined accordingly. In this implementation, the input images are twice as large as the output region.

After these modifications, recall that Horn's height and gradient scheme minimized the following error function (before linearization of the reflectance map)

$$\begin{aligned}&\iint [(E(x,y)-R(p,q))^2 \\ &\qquad + \lambda(p_x^2+p_y^2+q_x^2+q_y^2)+\mu((z_x-p)^2+(z_y-q)^2)]dxdy\end{aligned} \tag{3-3}$$

When using two images for the input, we minimize instead

$$\begin{aligned}&\iint [(E_1(x,y)-R(p,q))^2+(E_2(x,y)-R(p,q))^2 \\ &\qquad + \lambda(p_x^2+p_y^2+q_x^2+q_y^2)+\mu((z_x-p)^2+(z_y-q)^2)]\ dxdy\end{aligned} \tag{3-4}$$

Incorporating the linear approximation of R into equation (4) results in the following analogous iterative scheme (where $\lambda'' = \kappa\lambda' + \mu$) and $k = 10/3$

$$\begin{aligned}&D = \lambda''(\lambda''+R_p{}^2+R_q{}^2) \\ &A = \kappa\lambda'\delta\overline{p}_{kl}+\mu\delta z_x+(E_1-R)R_p+(E_2-R)R_p \\ &B = \kappa\lambda'\delta\overline{q}_{kl}+\mu\delta z_y+(E_1-R)R_q+(E_2-R)R_q \\ &\delta p_{kl} = (\lambda''+R_q{}^2)A-R_pR_qB\big/D \\ &\delta q_{kl} = (\lambda''+R_p{}^2)B-R_pR_qA\big/D\end{aligned} \tag{3-5}$$

where only the equations for A and B have changed from Horn's original scheme, and the new values for p, q, and z are still given by

$$\begin{aligned}p_{kl}^{(n+1)} &= p_0^{(n)}+\delta p_{kl}^{(n)} \\ q_{kl}^{(n+1)} &= q_0^{(n)}+\delta q_{kl}^{(n)}\end{aligned} \tag{3-6}$$

$$z_{kl}^{(n+1)} = \overline{z}_{kl}^{(n)} - \frac{\varepsilon^2}{\kappa}(p_x+q_y) \tag{3-7}$$

A minimum of the error function is achieved at the correct solution, unfortunately it may never be reached by this method, which in practice, performs only as well or worse than the monocular algorithm. There are two reasons for this; first of all, there is no explicit move to minimize the difference between E_1 and E_2 for a given x and y. This minimization is a variation of gray-level matching [Horn 1986] in the stereo correspondence problem. Specifically, we seek to also eliminate the stereo error

$$\iint_{xy} (E_1(x, y) - E_2(x, y))^2 dxdy \qquad (3\text{-}8)$$

keeping in mind that E is also dependent upon z from equation (3-2).

It might not be possible to completely minimize (eq. 3-8) due to aliasing of the images, but unless this is attempted, the poor terrain height estimate will cause pixels to be incorrectly matched across the left and right image. That is the second major problem with the simple binocular plan, as it introduces noise into the input. The iterative scheme considers a brightness component from each image for a particular spot. If it gets one of them from the wrong spot, some information is lost when components are added together. In the monocular case, a bad z estimate will distort the solution slightly, but no information is lost, and the solution becomes un-warped as it progresses. The two-image pixel mismatches are not as much of a problem initially, since iteration is started with a relatively high , which has a smoothing effect on the solution anyway. But unless the algorithm can get close to the solution and reduce the stereo error, it will not be possible to later resolve any smaller details that might still be blurred out from incorrect combination of the images.

3.3 Reducing the Stereo Error

An attempt must be made to aggressively reduce the stereo error before details can be refined using shape-from-shading information. One possibility is to extend the variational formulation to account for the change in image intensity with respect to z, and to include the stereo error in the functional to be minimized. A problem here is that the brightness gradient can draw the disparity in the wrong direction or cause it to become locked in a local false match [11]. This is most troublesome when the image varies quickly in gradient and the correct match is more than a few pixels away. We look instead for a method that can complement shape-from-shading and have the ability to break it out of local minima if necessary.

Consideration of the error term shows that it can be high at some terrain point for two reasons; first, because the z-value at that point is far enough from the correct value to cause a mismatch in the projection. Another possibility is that the corresponding point in one of the images is occluded, in which case there is not actually a match. This approach will not deal explicitly with occlusion since the terrain will be generally continuous and smooth enough to prevent this situation, which is more common in other scene types. On the other hand, when the stereo error is low, it can mean that the z is correct so the local match is good, or that the z might not be correct, but that the brightness of that region is uniform enough that the error is still low. High stereo errors will mostly occur when there is an edge running vertically through the image, for example, the top of a ridge, peak, or crater, or the bottom of a ravine that is facing the light source on one side, and facing away from it on the other. The stereo efforts should be focused on fixing areas where the error is high, because that means first of all, that something is wrong there, and second, that there are sufficient surface features to make a stereo analysis useful.

3.4 Stereo-based Warping

A simple method for further augmenting the height-and-gradient algorithm by stereo is now presented. The main goal of this augmentation is to make use of available stereo features, which provide sparse but absolute depth information, to guide the shape-from-shading approach to the best solution. Assuming that the monocular shape-from-shading was robust enough to find the correct relative solution, the second image could be used to place that solution at the correct absolute altitude, thereby arriving at the complete answer. Equation (8) could be easily modified to include the addition of a constant, allowing the entire solution to be shifted up or down as necessary without affecting the shape-from-shading progress at all. But we can do better than that, because the stereo information could also be used to speed up the convergence process. It has been empirically found that the height-and-gradient solution-in-progress can be locally adjusted one way or another, without de-stabilizing the convergence. Additionally, the use of local-z-averaging in the algorithm should allow these local adjustments to propagate outward to the surrounding areas. In other words, stereo information can be used to push the elevation of some feature's region toward its true height, while the height-and-gradient process takes care of properly aligning the adjacent regions by filling in the sparse stereo data with shading.

The stereo error is sampled every 10 pixels each direction on a square grid. Each sample point is a potential location for z-adjustment. The sampling is done for one point every other iteration of the height-and-gradient loop, left-right, top-bottom until all 100 points have been sampled, then the grid shifts right/down 1 pixel and repeats. This continues until every point has been sampled, then it starts from the beginning. The reason for the shift is to reduce the chance of repetitive errors.

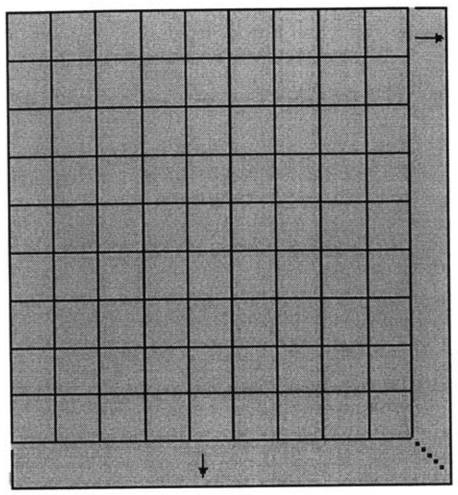


Figure 3-2: The shifting error-sample grid over the image

The warp process takes place in three steps:

- First, when a point *(sx,sy)* is "sampled," the local stereo error (equation 9) is calculated for the 9x9 pixel region surrounding that point. If the error is above a threshold value τ , then we try to improve that error (step 2). Otherwise, skip to the next sample.
- To find the best improvement for a region, test what z-shift (at intervals of ~0.5) minimizes the current local stereo error. That value is *W*, the best warp. The convenience of this test is that we do not try to assign a uniform depth to the entire region, but instead only check if the current shape should be raised or lowered. Because shape-from-shading also affects the region, any foreshortening problems in correlation are eventually worked out.
- Finally, additively adjust the local z values towards *W* using an error distribution function. The adjustment value at any point (k,l) is given by

$$\alpha W e^{-((sx-k)^2+(sy-l)^2)/\beta} \tag{10}$$

where and are constants that determine the magnitude and width of the displacement, respectively.

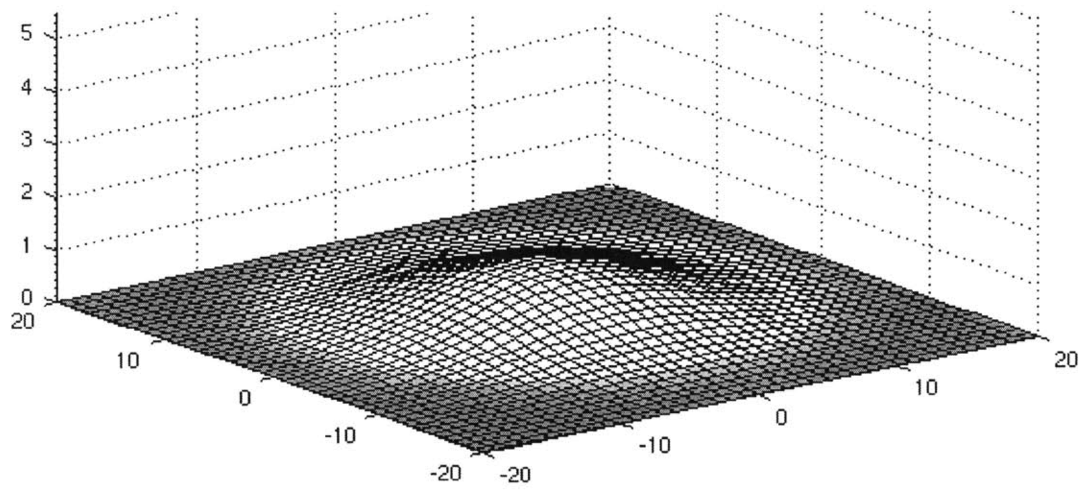


Figure 3-3: An example warp with *α=W=1* and *β=100.*

3.5 Considerations

When selecting values for the constants and , several factors must be taken into consideration. Alpha controls the magnitude of the displacement, and should be large enough to effectively move the solution, but not so large that it completely distorts it. Alpha should start relatively large, and be reduced as the stereo error decreases. It should generally be less than 1, so that it takes a few iterations for the region to ease into the correct height. This also minimizes the harmful effects that one bad match can have. Values greater than 1 tend to cause instability in the iteration. determines the radius of the disturbance, and should be chosen carefully to work well with the height-and-gradient process. It should be wide enough that the entire local region is adjusted, but not so coarse that it extends into dissimilar regions. Furthermore, when is high, the shape-from-shading recovers quickly from the warp since the solution tends toward smoothness.

Later in the process when the departure-from-smoothness term is reduced, large warps are not so rapidly diffused, so they can actually undo progress that has been made by the shading algorithm. In other words, the width of the warp should decrease along with the smoothness constraint so as not to hinder the convergence.

Another important consideration is the threshold value , which determines whether the local stereo error is large enough to require some correction. The threshold should ideally be chosen such that an error value greater than represents useful stereo information, and anything less is noise or weak stereo information. It might seem like a good idea to reduce this threshold as the solution is refined, in order to completely minimize the stereo error. Counter-intuitively, it is actually better to increase , recalling that the fine details are shape-from-shading's specialty. The stereo warp is given more weight initially, as it helps speed convergence from the initial conditions to a rough solution. It does this at the expense of potentially increasing the gradient error, and the height-and-gradient process must catch up after each warp. The progressive decrease in reduces the ability to "catch up" later in the resolution, and so disturbances at this stage should be avoided if possible, even if the stereo error begins to increase somewhat. We would rather have a map that shows the land forms correctly with the possibility of some small error in absolute altitude, than a map that shows the altitudes exactly at sparse stereo features and potentially misrepresents the appearance of the surrounding terrain. As a result, shape-from-shading is weighted much more heavily near the end of the process, which ensures that we don't accidentally add incorrectly formed "features" through aberrant warping.

4 Results

4.1 Synthetic Input Data

The algorithm is tested using synthetic image data, so that the exact solution is known for comparison, and the camera geometry can be easily controlled. This allows easy evaluation of the method on a variety of terrains, and avoids some potential problems with real images that are beyond the scope of this work. The images are designed to simulate the qualities that might be found in real images taken from an orbital survey of a barren planet. The terrain surface itself should have constant albedo (no color change independent of lighting conditions) and a Lambertian reflectance map producing gray-scale images. The terrain will be smooth in some places and contain features or surface texture in other places.

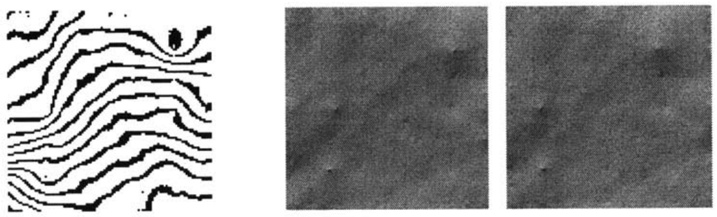


Figure 4-1: Contours of terrain #1234 and associated stereo image pair

In order to create interesting yet random terrain, we use the diamond-square terrain fractal [12]. The randomness is controlled by specifying some seed to the number generator, so that the same pseudo-random terrain is created every time a particular seed is used. Figure 4-1 above is an example of a random fractal terrain. White background in the contour diagram indicates that the elevation is above the sea-level reference. Gray coloring would indicate a negative elevation. This terrain, for example, is entirely positive, and slopes downward towards the bottom of the image. The images are "taken" at a resolution of 1 unit/pixel, from cameras setup at height a height of 100 units, with a 100 unit baseline. The stereo pair shows only the center 100x100 pixels of each image, but they are both actually 200x200 in size, in order to guarantee a brightness value can be projected from each point in the reconstructed region (100x100 units). The b/h ratio of 1.0 gives relatively strong stereo coverage, and in the optimal case of distinctly textured images, we could expect this to yield stereo height accuracy of large features on the order of 0.5-1.0 pixels or 0.5-1.0 units. The light source is toward the northwest $[-.5 \quad 1 \quad 1]^T$ at an infinite distance.

Ideally, we should generate the images at some higher resolution, and then use a block averaging technique to reduce them. This would create the cleanest quality input for testing. However, since we are generating the images again each session, we use only single-value samples to create the images, in order to keep the computational time manageable. As a result, there is a small amount of noise in the images, but that helps to simulate the conditions of real images, which should be the eventual goal in such an investigation.

4.2 Quantitative Measures of Correctness

In order to gauge the state of the solution at any stage during iteration, we use various quantitative measures. The first is brightness error in p and q (equation 2-3). This value becomes low very fast. A better indicator is the brightness error in gradient

$$\iint (E(x,y) - R(z_x, z_y))^2 \, dx\,dy \tag{4-1}$$

which better captures the state of the z solution from the shape-from-shading perspective. A third error measure is the stereo error (equation 3-8) shows the discrepancy remaining between the images in the current solution heights. These three indicators are the values this scheme works to minimize, but it is also instructive to measure how close this intended minimization is bringing us to the ground truth solution data. Recalling that we are looking for a combination of correct elevation data and correct shape resolution two additional error measures are introduced. First is the straightforward average squared elevation error

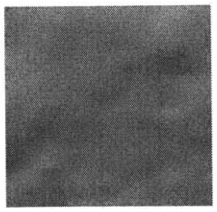
$$\iint \frac{(z-z')^2}{10000}\,dx\,dy \qquad \text{where } z' \text{ is the ground truth data} \tag{4-2}$$

and second is the error in brightness from gradient using an alternate light source (reflectance map) to shade both the current and ground truth solutions.

$$\iint (E_2(x,y) - R_2(z_x, z_y))^2\,dx\,dy \tag{4-3}$$

The last error measure seems to be the most difficult to reduce, the algorithm is only indirectly minimizing this sum. These two error measures should not be used to affect the solution in any way, since they would not be available unless it was already known.

4.3 Performance of Binocular shading alone

a) 

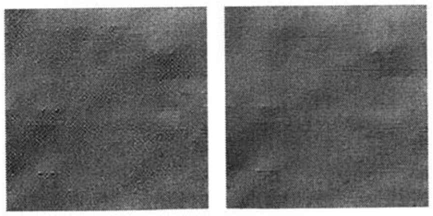
b) 

Figure 4-2: Shape-from-shading without adequate reduction of the stereo error

The effects of high stereo error are shown in Figure 4-2. By image (a), it is evident early in the iteration that the stereo is incorrectly matched. Notice the misalignment of point features is somewhat hidden by the smoothing effect. The alignment cannot be corrected, so when the smoothing term is later reduced (4-2b), the duplication is clearly visible.

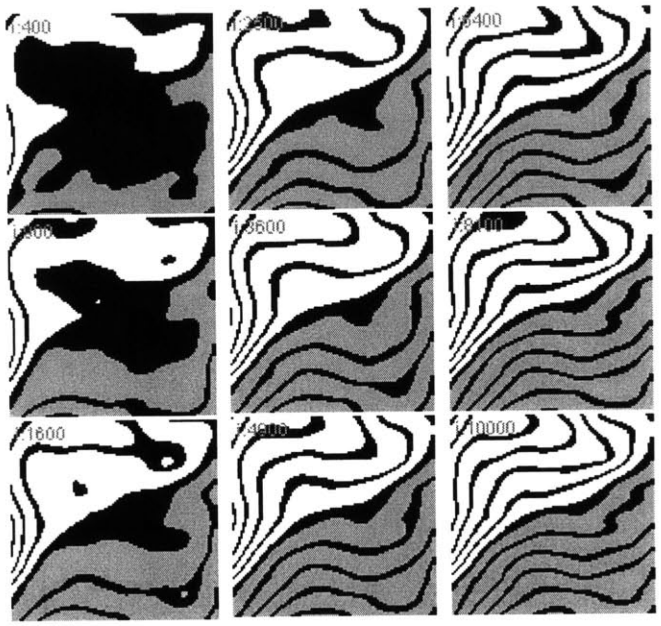


Figure 4-3: Convergence to an incorrect solution

4.4 Augmented Binocular Shading

Figure 4-4 shows the progress of a solution using the augmented scheme in a shaded and contour view. Discernable points in the images become aligned early in the scheme, as a result of the coarse z-adjustments between shape-from-shading iterations. At some stages, the gradient may be quite wrong, but this is quickly corrected as the iterations continue. The large dents visible in the early frames are caused by the warping process, specifically when the stereo error is initially large (and   are then also large). After the stereo error is reduced, the height-and-gradient takes care of filling in the details of the surface. The result is quite close to the ground truth height data. The average elevation error for this input converges to just below 1 unit/pixel, or right in the best possible range, considering the resolution of the stereo data. For comparison, a disparity map generated by Zitnick and Kanade's algorithm [17] is shown for the left image in figure 4-5. At the same time, dents in the surface as small as 1-2 units can be seen reconstructed in the DEM in figure 4-6, which was not possible with stereo alone. Without optimization, the execution time on a 1.4GHz AMD Athlon machine running Windows XP is approximately 25 minutes for 10,000 iterations of the algorithm.

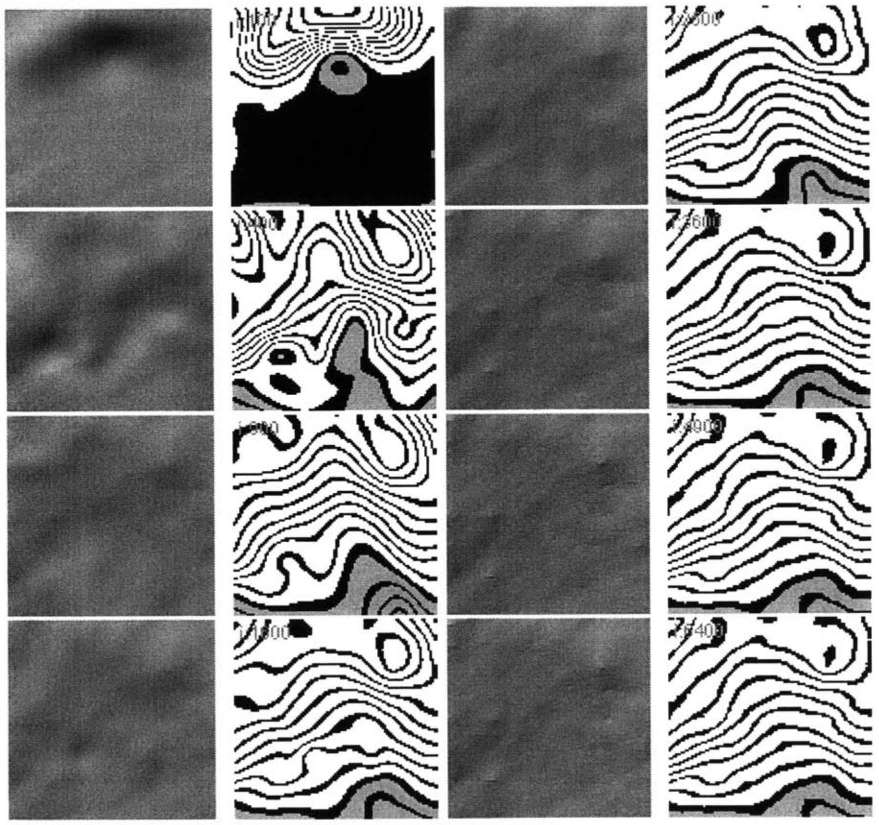


Figure 4-4: Improved convergence by active minimization of stereo error

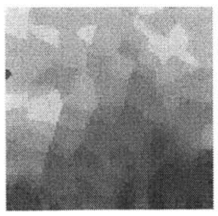


Figure 4-5: Disparity map generated by a stereo-only matching algorithm [17]

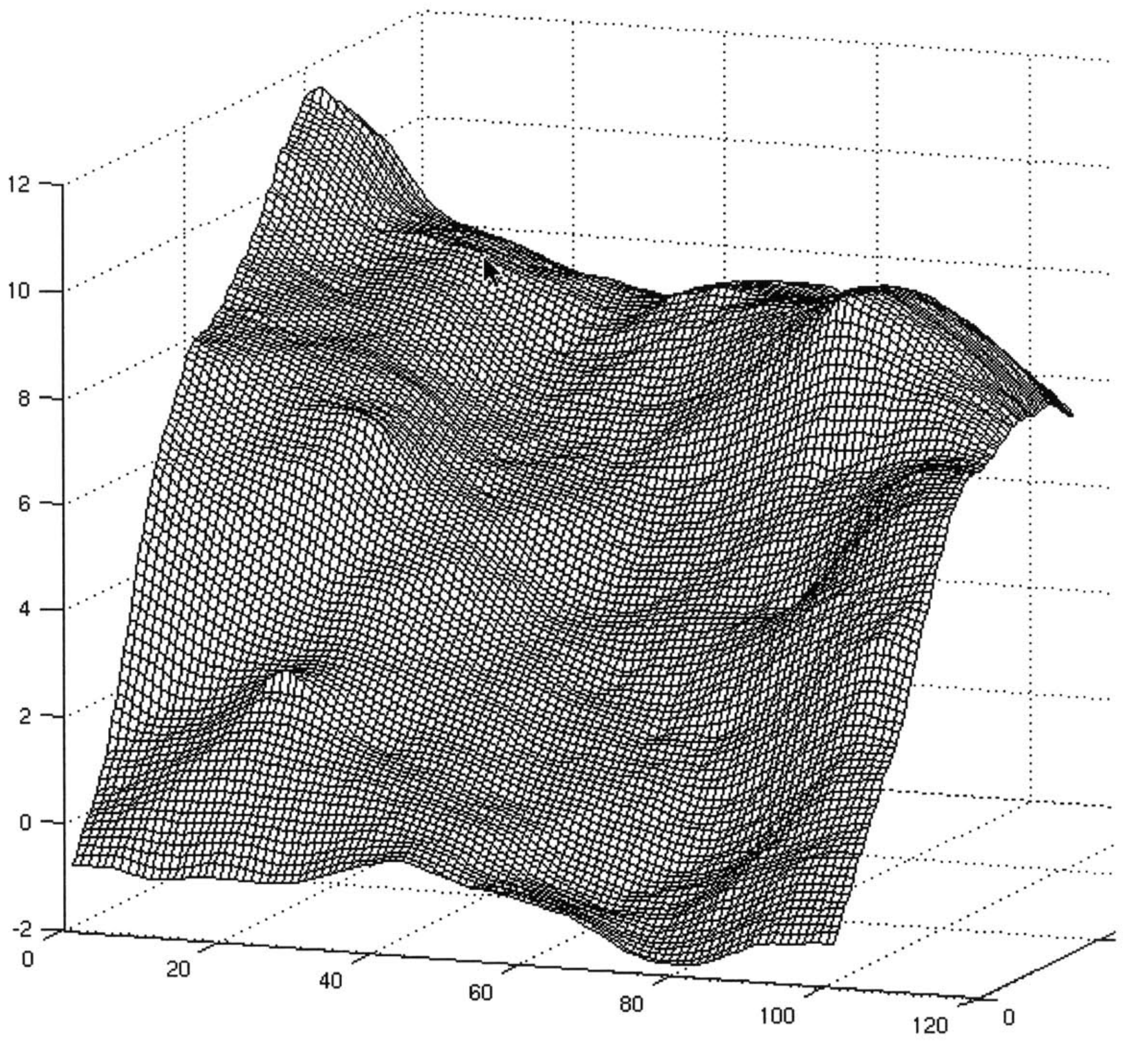
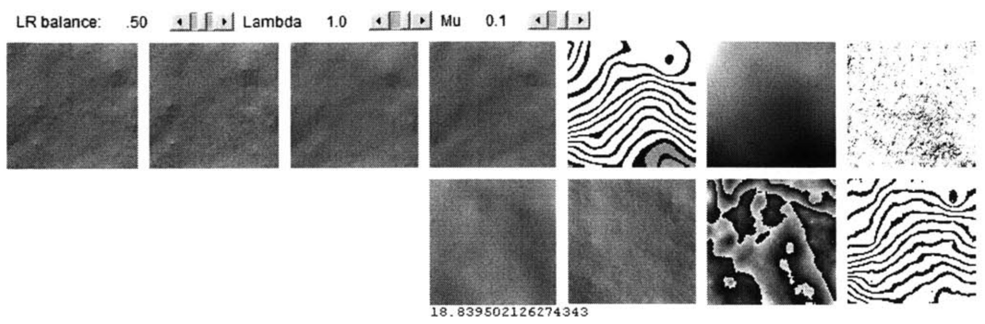


Figure 4-6: Generated DEM Solution with ~8x vertical exaggeration

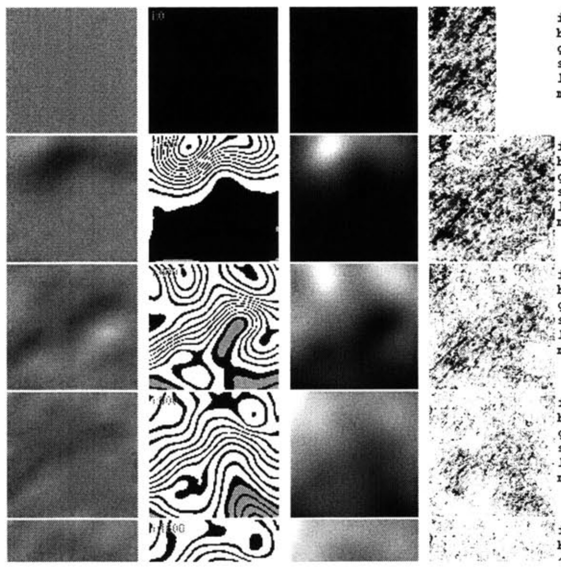
1) Input L	2) Input R	3) reconstructed from slope (p&q)	4) reconstr. from gradient (dz)	5) elevation contours (gray=negative, interval=1)	6) elevation color map (dark=lower, light=higher)	11) stereo error (red/blue indicates sign of error)
			8) #4 viewed with an alternate light source	*9) reference solution (what #8 should look like)*	7) absolute error (mod 1.0)	*10) reference contours (interval=1)*



```
LR balance:  .50   Lambda  1.0   Mu  0.1
```

```
18.839502126274343
```

```
i: 1600 Br.e: 2.5471475707910596 Gr.e: 6.179383844739583 St.e: 2.868471704525528 int.e: 6.37554109766342 Abs.e: 1.9248738398132
i: 1650 Br.e: 2.545357225838227З Gr.e: 5.826644407935433 St.e: 2.825401443239434 int.e: 5.166147248214949 Abs.e: 1.912493552615
i: 1700 Br.e: 2.55840951462558Z Gr.e: 5.541165918284692 St.e: 2.6967781568549953 int.e: 4.580848272201715 Abs.e: 1.766534767230
i: 1750 Br.e: 2.6039503723677746 Gr.e: 6.2954115615274375 St.e: 2.512697721957338 int.e: 5.825754111982523 Abs.e: 1.71134940294
i: 1800 Br.e: 2.570135696563505 Gr.e: 5.372084265240563 St.e: 2.600872600257081 int.e: 5.126697649500798 Abs.e: 1.7916596574476
i: 1850 Br.e: 2.568294976329968 Gr.e: 5.290018389401876 St.e: 2.5758620944844237 int.e: 5.040712161321374 Abs.e: 1.826474431492
```



```
i:0
br.e:80.491580
gr.e:80.491580
st.e:18.044789
lambda:1.0
mu:0.1

i:100
br.e:3.2575821
gr.e:81.372799
st.e:10.123446
lambda:1.0
mu:0.1

i:400
br.e:2.8530241
gr.e:48.658095
st.e:6.5006417
lambda:1.0
mu:0.1

i:900
br.e:2.5676707
gr.e:20.988243
st.e:4.1600494
lambda:1.0
mu:0.1

i:1600
br.e:2.5471475
```

Figure 4-7: Progress display of the stereo height and gradient applet

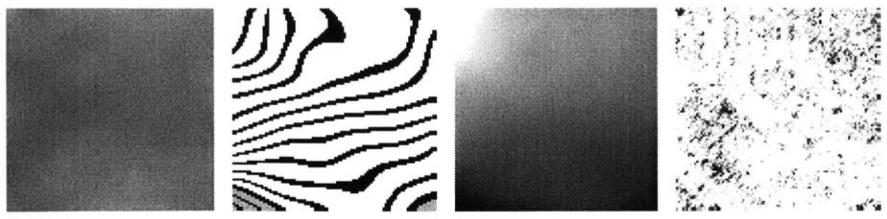


Figure 4-8: Solution component from stereo only (no shading)

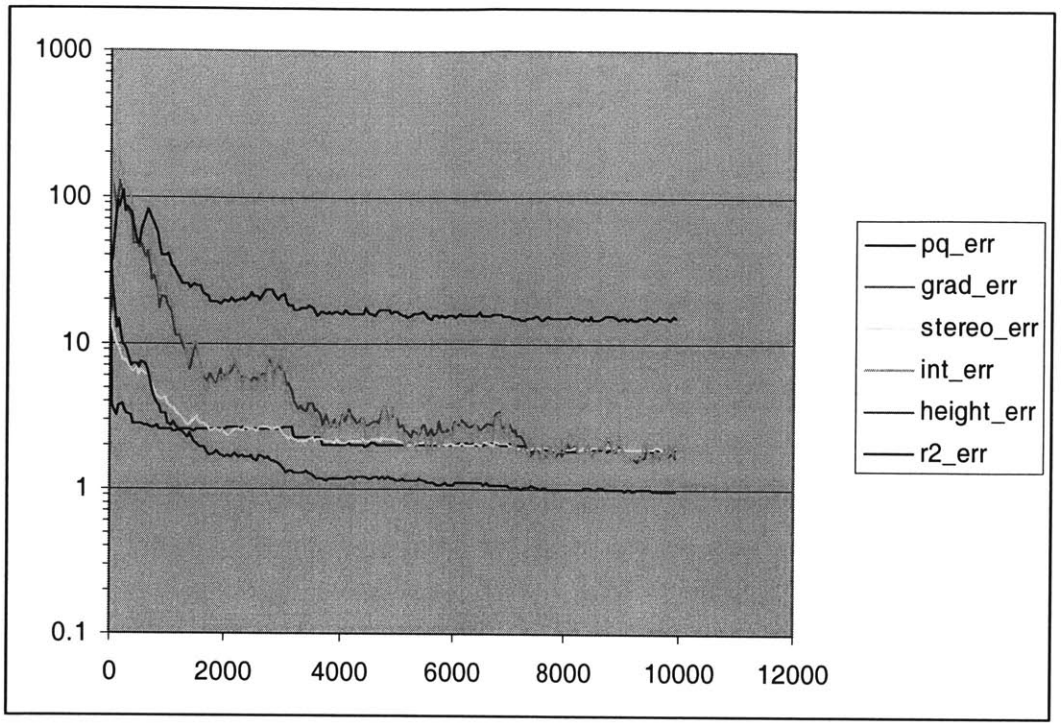


Figure 4-9: Convergence of various error measures

It is interesting to notice that all 4 intensity error measures eventually converge to approximately the same non-zero value. This is a good indication of the level of noise in the input images. If the regularizing constraints are reduced any further, the solution begins to become unstable and diverges from the correct answer.

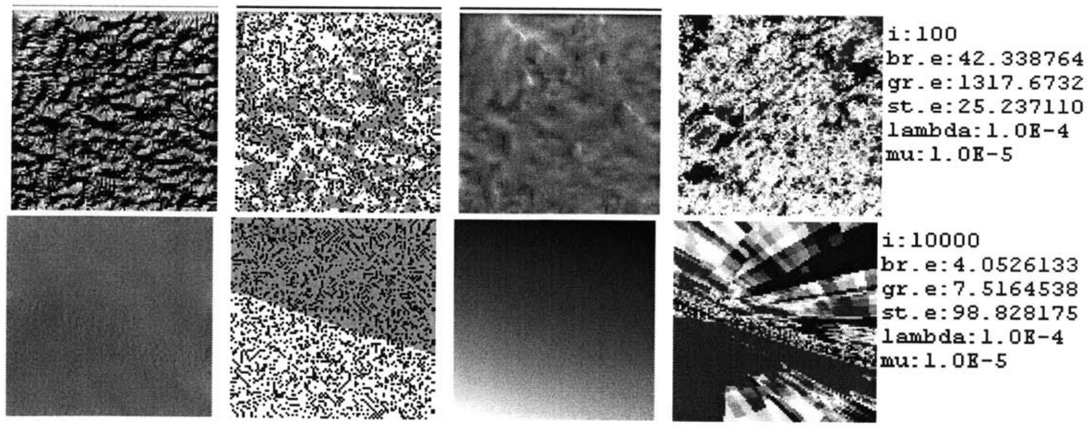


Figure 4-10: Divergent Solution

5 Conclusions

This combined method successfully integrates the height-and-gradient method and shape-from-stereo, using an aggressive warping process to adjust the shape-from-shading solution at features where there is a relatively high stereo error between the two images. The images are not highly textured, so dependable stereo data is sparse. However, because shape-from-shading works in these areas between features, the resulting elevation map is as accurate as if there were dense texture, and as precise in resolving small features as the height-and-gradient method but with faster convergence. In further investigation, this method could be extended to use other types of stereo matching, different schemes for warping, or become more tightly integrated with the height and gradient scheme.

6 References

[1] A. Blake, A. Zisserman, and G. Knowles. Surface descriptions from stereo and shading. *Image and Vision Computing,* 3:183-191, 1985.

[2] R. Butterfield, R. M. Harkness, and K. Z. Andrawes. A stereo-photogrammetric method for measuring displacement fields. *Geotechnique*, **20**, 308-314, (1970).

[3] M. T. Chiaradia, A. Distante, and E. Stella Three-dimensional surface reconstruction integrating shading and sparse stereo data. *Optical Engineering,* 28(9):935-942, September 1989.

[4] J. E. Cryer, P. Tsai, and M. Shah. Integration of Shape from Shading and Stereo. *Pattern Recognition,* Vol 28, No. 7, pp. 1033-1043, 1995.

[5] E. Dorrer, X. Zhou. Towards Optimal Relief Representation From Mars Imagery By Combination Of DEM And Shape-From-Shading. *Int. Arch. Photogrammetry & Remote Sensing,* Vol 32, Part 4, Stuttgart 1998, 156-161, 1998.

[6] P. Fua and Y.G Leclerc. Using three-dimensional meshes to combine image-based and geometry-based constraints. *Proc. 3rd European Conference on Computer Vision,* pp. 282-291, May 1994.

[7] C. F. Hall and E. L. Hall. A nonlinear model for the spatial characteristics of the human visual system. *IEEE Trans. System. Man. Cybern.* 6, 161-170, 1977.

[8] D.R. Hougen and N. Ahuja. Estimation of the light source distribution and its use in integrated shape recovery from stereo and shading. *Proc. 4th International Conference on Computer Vision,* pp. 148-155, May 1993.

[9] B.K.P. Horn. Height and Gradient from Shading. *International Journal of Computer Vision,* Volume 5, number 1, 37-75, The Netherlands, 1990.

[10] B.K.P. Horn, R.S. Szeliski, & A.L. Yuille. Impossible Shaded Images, *IEEE Transactions of Pattern Analysis and Machine Intelligence*, February 1993, Vol. 15, No. 2, pp. 166--170.

[11] B.K.P. Horn. Robot Vision. MIT Press: Cambridge, MA; and McGraw-Hill: New York, 1986.

[12] Gavin S. P. Miller. The Definition and Rendering of Terrain Maps. *Computer Graphics*, vol. 20, no. 4, pp 9-48, 1986.

[13] A. P. Pentland. Local Shading Analysis. *IEEE Transactions of Pattern Analysis and Machine Intelligence,* 6(2), 170-187, 1987.

[14] T. Poggio, E. B. Gamble, J. J. Little. *Science,* New Series, Volume 242, Issue 4877 (Oct. 21 1988), 436-440.

[15] C. Sun. Fast Stereo Matching Using Rectangular Subregioning and 3D Maximum-Surface Techniques. *International Journal of Computer Vision.* 47(1/2/3):99-117, May 2002.

[16] J. Y. Zheng and F. Kishino. Verifying and combining different visual cues into a 3D model. *Proc. IEEE Conf. Computer Vision and Pattern Recognition, 1992.* pp 777-780, June 1992.

[17] C. Zitnick and T. Kanade. A Cooperative Algorithm for Stereo Matching and Occlusion Detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 22(7): 675-684, 2000

7 Appendix A: Java Source Code

This section presents the java (jdk1.4.0) source code which implements the combined stereo shape-from-shading algorithm. The compiled applet is also available for viewing in a web browser here:

http://framework2.lcs.mit.edu/thesis/HGApplet.html

or permanently:

http://thesis.attackpoint.org

Class Overview:
HG- the main UI component that performs the iteration and displays the result
Terrain- a test terrain that uses a mathematical function z(x,y) for the surface
FractalTerrain- an extension of Terrain that uses a fractal surface instead
Camera- generates the various images of the terrain

```
/*
 * HG.java
 *
 * Created on April 11, 2002, 4:53 PM
 */
import java.awt.*;
import java.awt.image.*;
import java.util.*;
/**
 *
 * @author  kwalker
 * @version
 */
public class HG extends Canvas implements Runnable {

    double SPACING=1;
    int M=100;
    int N=100;
    double E0=1;
    long seed=1234;

    double[][] z=new double[101][101],z2=new double[101][101];
    double[][] p=new double[100][100],p2=new double[100][100];
    double[][] q=new double[100][100],q2=new double[100][100];
    double[][] e=new double[200][200];
    double[][] e1=new double[200][200];
    double[][] e2=new double[200][200];
    double[][] tempf=new double[100][100];
    double[][] tempz=new double[101][101];
    double[][] err=new double[101][101];
    //double[][] e1b=new double[200][200];
    double[][] e2b=new double[200][200];
    double[][] contours;
    double[][] se=new double[101][101];


    Terrain terrain;
    Thread thread;
    String msg="";
    Vector msgs=new Vector();
    BufferedImage bi=new BufferedImage(1000,300,BufferedImage.TYPE_INT_RGB);
    BufferedImage progress=new
BufferedImage(530,1500,BufferedImage.TYPE_INT_RGB);
    boolean bi_init=false;

    /** Creates new HG */
    public HG() {
        super();
        thread=new Thread(this);
        thread.start();
        this.setSize(1000,1800);
        progress.getGraphics().fillRect(0,0,600,2000);
        bi.getGraphics().fillRect(0,0,1000,350);

    }

    int pc=0;
    double err2=0;

    public void paint(Graphics g2){
        //System.out.println("painting");
        Graphics g=bi.getGraphics();
        g.fillRect(0,210,1000,90);
        g.setColor(Color.black);
                g.setFont(new Font("Courier",Font.PLAIN,11));
        try {
        g.drawString(msgs.elementAt(0).toString(),10,290);
        g.drawString(msgs.elementAt(1).toString(),10,280);
        g.drawString(msgs.elementAt(2).toString(),10,270);
        g.drawString(msgs.elementAt(3).toString(),10,260);
        g.drawString(msgs.elementAt(4).toString(),10,250);
        g.drawString(msgs.elementAt(5).toString(),10,240);
        } catch (Exception e) { }

        float hmin=0,hmax=(float)0.01;
        for (int x=0;x<100;x++){
            for (int y=0;y<100;y++){
                float mval=(float)(z[x][y]);
                if (mval < hmin) hmin=mval;
                if (mval > hmax) hmax=mval;
            }
        }
        err2=0;
        for (int x=0;x<100;x++){
            for (int y=0;y<100;y++){
                if (pc<2) {
                    float e1xy=(float)e1[x+50][y+50];
                    g.setColor(new Color(e1xy,e1xy,e1xy));
                    g.drawRect(x,y,1,1);
                    float e2xy=(float)e2[x+50][y+50];
                    g.setColor(new Color(e2xy,e2xy,e2xy));
                    g.drawRect(x+110,y,1,1);

                //float e1bxy=(float)e1b[x+50][y+50];
                //g.setColor(new Color(e1bxy,e1bxy,e1bxy));
                //g.drawRect(x,y+110,1,1);
                    float e2bxy=(float)e2b[x+50][y+50];
                    g.setColor(new Color(e2bxy,e2bxy,e2bxy));
                    g.drawRect(x+440,y+110,1,1);

                    float cv=(float)contours[x][y];
                    g.setColor(new Color(cv,cv,cv));
                    g.drawRect(x+660,y+110,1,1);

                }

                float rval=(float)R(p[x][y],q[x][y]);
                g.setColor(new Color(rval,rval,rval));
                g.drawRect(x+220,y,1,1);

                float gval=(float)R(Zx(x,y),Zy(x,y));
                g.setColor(new Color(gval,gval,gval));
                g.drawRect(x+330,y,1,1);

                float gvalb=(float)terrain.R2(Zx(x,y),Zy(x,y));
                g.setColor(new Color(gvalb,gvalb,gvalb));
                g.drawRect(x+330,y+110,1,1);

                float zval=1;
                if (Math.abs(z[x][y])%1<.25) {
                    zval=(float)0;
                }
                else if (z[x][y]<0) zval=(float).75;
                g.setColor(new Color(zval,zval,zval));
                g.drawRect(x+440,y,1,1);

                // normalized height map
                try {
                float mval=(float)(z[x][y]);
                mval=(mval-hmin)/(hmax-hmin);
                g.setColor(new Color(mval,mval,mval));
                g.drawRect(x+550,y,1,1);
                } catch (Exception e) {
                    //g.setColor(Color.black);
                    //g.drawString(e.getMessage()+"   ",550,200);
                }

                float ev=(float)(err[x][y]/1.);
                if (ev>0){
                    g.setColor(new Color(1,(float)(1-(ev%1.0)),(float)(1-
(ev%1.0))));
                } else {
                    g.setColor(new Color((float)(1-(-ev)%1.0),(float)(1-(-
ev)%1.0),1));
                }
                g.drawRect(x+550,y+110,1,1);

                // stereo disparity map
                float dval=1-(float)disparity[x][y]/(float)DEPTH_RANGE;
                g.setColor(new Color(dval,dval,dval));
                g.drawRect(x+770,y+0,1,1);


                float sev=((float)se[x][y]*20);
                if (sev>0){
                    if (sev>1) sev=1;
                    g.setColor(new Color(1,1-(float)sqr(sev),1-
(float)sqr(sev)));
```

```
                } else {
                    if (sev<-1) sev=-1;
                    g.setColor(new Color(1-(float)sqr(sev),1-
(float)sqr(sev),1));
                }
                //g.setColor(new Color(sev,sev,sev));
                g.drawRect(x+660,y,1,1);
                err2+=Math.pow(gvalb-(float)e2b[x+50][y+50],2);
            }
        }

        g.setColor(Color.red);
        g.drawString(err2+"         ",330,220);

        g2.drawImage(bi,0,0,null);
        g2.drawImage(progress,0,350,null);
    }


    // special access methods to maintain boundary conditions
    // gradients

    double get_p(int x, int y){
        if (x<0) x=0;
        else if (x>M-1) x=M-1;
        if (y<0) y=0;
        else if (y>N-1) y=N-1;
        return p[x][y];
    }

    double get_q(int x, int y){
        if (x<0) x=0;
        else if (x>M-1) x=M-1;
        if (y<0) y=0;
        else if (y>N-1) y=N-1;
        return q[x][y];
    }

    double Px(int k, int l){
        return (1/(2*SPACING))*(get_p(k,l)-get_p(k-1,l)+get_p(k,l-1)-get_p(k-
1,l-1));
    }

    double Qy(int k, int l){
        return (1/(2*SPACING))*(get_q(k,l)-get_q(k,l-1)+get_q(k-1,l)-get_q(k-
1,l-1));
    }
    // reflectance map function
    double R(double p, double q) {
        return terrain.R(p,q);
    }

    double delta=0.01;
    double Rp(double p,double q){
        return ( R(p+delta,q) - R(p-delta,q) )/ (2*delta);
    }
    double Rq(double p,double q){
        return ( R(p,q+delta) - R(p,q-delta) )/ (2*delta);
    }
    double Zx(int k, int l){
        return (1/(2*SPACING))*(z[k+1][l]-z[k][l]+z[k+1][l+1]-z[k][l+1]);
    }
    double Zy(int k, int l){
        return (1/(2*SPACING))*(z[k][l+1]-z[k][l]+z[k+1][l+1]-z[k+1][l]);
    }

    double get_z(int x, int y){
        int pm=0,qm=0,xa=0,ya=0;
        if (x<0)       { x=0; pm=-1; }
        else if (x>M)  { x=M; pm= 1; xa=-1; }
        if (y<0)       { y=0; qm=-1; }
        else if (y>N)  { y=N; qm= 1; ya=-1; }
        if (pm==0 && qm==0) return z[x][y];
        else return z[x][y] + (pm * SPACING / 2 * (get_p(x+xa,y) +
get_p(x+xa,y-1)))
        + (qm * SPACING / 2 * (get_q(x,y+ya) + get_q(x-1,y+ya)));

    }


    // local average functions
    double Zbar(int k,int l){
        return (.2*(get_z(k-1,l)+get_z(k+1,l)+get_z(k,l-1)+get_z(k,l+1))
        +.05*(get_z(k-1,l-1)+get_z(k+1,l-1)+get_z(k-1,l+1)+get_z(k+1,l+1)));
    }
    double Pbar(int k,int l){
        return (.2*(get_p(k-1,l)+get_p(k+1,l)+get_p(k,l-1)+get_p(k,l+1))
        +.05*(get_p(k-1,l-1)+get_p(k+1,l-1)+get_p(k-1,l+1)+get_p(k+1,l+1)));
    }
    double Qbar(int k,int l){
        return (.2*(get_q(k-1,l)+get_q(k+1,l)+get_q(k,l-1)+get_q(k,l+1))
        +.05*(get_q(k-1,l-1)+get_q(k+1,l-1)+get_q(k-1,l+1)+get_q(k+1,l+1)));
    }

    double sqr(double x){
        return x*x;
    }

    // get e for a point in the terrain, adjusting for perspective projection
    double get_e_prsp(int xa, int ya){
        Point3D camera_rel=new Point3D(0,50,camera_z);
        Point3D ray = new Point3D(xa-camera_rel.x,ya-camera_rel.y,camera_rel.z-
Zbar(xa,ya));

        double zscale=camera_rel.z/ray.z;
        ray.x*=zscale;
        ray.y*=zscale;

        //check e array bounds
        int xb=(int)(ray.x+camera_rel.x+M/2);
        int yb=(int)(ray.y+camera_rel.y+N/2);

        if (xb<0) xb=0;
        else if (xb>M*2-1) xb=M*2-1;
        if (yb<0) yb=0;
        else if (yb>N*2-1) yb=N*2-1;


        return e[xb][yb];
    }

    double get_e1_prsp(int xa, int ya){
        Point3D camera_rel=new Point3D(0,50,camera_z);
        Point3D ray = new Point3D(xa-camera_rel.x,ya-camera_rel.y,camera_rel.z-
Zbar(xa,ya));
```

```
        double zscale=camera_rel.z/ray.z;
        ray.x*=zscale;
        ray.y*=zscale;

        //check e array bounds
        int xb=(int)(ray.x+camera_rel.x+M/2);
        int yb=(int)(ray.y+camera_rel.y+N/2);

        if (xb<0) xb=0;
        else if (xb>M*2-1) xb=M*2-1;
        if (yb<0) yb=0;
        else if (yb>N*2-1) yb=N*2-1;


        return e1[xb][yb];
    }

    double get_e2_prsp(int xa, int ya){
        Point3D camera_rel=new Point3D(100,50,camera_z);
        Point3D ray = new Point3D(xa-camera_rel.x,ya-camera_rel.y,camera_rel.z-
Zbar(xa,ya));

        double zscale=camera_rel.z/ray.z;
        ray.x*=zscale;
        ray.y*=zscale;

        //check e array bounds
        int xb=(int)(ray.x+camera_rel.x+M/2);
        int yb=(int)(ray.y+camera_rel.y+N/2);

        if (xb<0) xb=0;
        else if (xb>M*2-1) xb=M*2-1;
        if (yb<0) yb=0;
        else if (yb>N*2-1) yb=N*2-1;


        return e2[xb][yb];
    }

    double get_e1_prsp(int xa, int ya, double dz){
        Point3D camera_rel=new Point3D(0,50,camera_z);
        Point3D ray = new Point3D(xa-camera_rel.x,ya-camera_rel.y,camera_rel.z-
Zbar(xa,ya)-dz);
        double zscale=camera_rel.z/ray.z;
        ray.x*=zscale;
        ray.y*=zscale;
        //check e array bounds
        int xb=(int)(ray.x+camera_rel.x+M/2);
        int yb=(int)(ray.y+camera_rel.y+N/2);
        if (xb<0) xb=0;
        else if (xb>M*2-1) xb=M*2-1;
        if (yb<0) yb=0;
        else if (yb>N*2-1) yb=N*2-1;
        return e1[xb][yb];
    }

    double get_e2_prsp(int xa, int ya, double dz){
        Point3D camera_rel=new Point3D(100,50,camera_z);
        Point3D ray = new Point3D(xa-camera_rel.x,ya-camera_rel.y,camera_rel.z-
Zbar(xa,ya)-dz);
        double zscale=camera_rel.z/ray.z;
        ray.x*=zscale;
    ray.y*=zscale;
    //check e array bounds
    int xb=(int)(ray.x+camera_rel.x+M/2);
    int yb=(int)(ray.y+camera_rel.y+N/2);
    if (xb<0) xb=0;
    else if (xb>M*2-1) xb=M*2-1;
    if (yb<0) yb=0;
    else if (yb>N*2-1) yb=N*2-1;
    return e2[xb][yb];
}

public double getStereoError(int x, int y, int d, double dz){
    double err=0;
    for (int i=x-d;i<=x+d;i++){
        for (int j=y-d;j<=y+d;j++){
            err+=Math.abs(get_e1_prsp(x,y,dz)-get_e2_prsp(x,y,dz));
        }
    }
    return err;
}

public double getBestWarp(int x, int y){
    double range=10.0;
    double step=0.2;
    int d=5;
    double minErr=getStereoError(x,y,d,0.0);
    double bestWarp=0.0;
    for (double warp=-range;warp<=range;warp+=step){
        double err=getStereoError(x,y,d,warp);
        //System.out.println("p:"+warp+" e:"+err);
        if (err<minErr){
            minErr=err;
            bestWarp=warp;
        }
    }
    return bestWarp;
}

// "constants"
public double mu=.1;
public double lambda=1;
public double lrbal=0.5;

void iterate() {

    double K=10.0/3.0;
    int di=50,pi=2,pic=0;
    double azx=0,azy=0,ap=0,aq=0;

    for(int i=0;i<=10000;i++){

        //may need to recalculate if l or m change
        double klp=K*lambda/sqr(SPACING);
        double lpp=klp+mu;

        double bright_e=0,grad_e=0,stereo_e=0,abs_e=0,alt_e=0,int_e=0;
        double max_se=0;
        int mse_x=0,mse_y=0;
        int mse_x2=0,mse_y2=0;

        for(int x=0;x<M;x++){
            for(int y=0;y<N;y++){
                    double p0=p[x][y];
                    double q0=q[x][y];

                    double pb=Pbar(x,y);
                    double qb=Qbar(x,y);
                    double rp=Rp(pb,qb);
                    double rq=Rq(pb,qb);

                    //double EmR=e[x][y]-R(pb,qb);
                    //double EmR=get_e_prsp(x,y)-R(pb,qb);
                    double EmR=(1-
lrbal)*get_e1_prsp(x,y)+lrbal*get_e2_prsp(x,y)-R(pb,qb);
                    //double EmR=(1-
lrbal)*Math.abs(get_e1_prsp(x,y))+lrbal*Math.abs(get_e2_prsp(x,y))-R(pb,qb);
                    double Ediff=get_e1_prsp(x,y)-get_e2_prsp(x,y);
                    se[x][y]=Ediff;
                    stereo_e+=sqr(Ediff);

                    if (i%pi==0) {
                        double f;
                        if ((f=getStereoError(x,y,2,0.))>max_se) {
                            max_se=f;
                            mse_x=x;
                            mse_y=y;
                        }
                    }

                    double D=lpp*(lpp+sqr(rp)+sqr(rq));
                    double A=klp*(pb-p0) + mu*(Zx(x,y)-p0) + EmR*rp;
                    double B=klp*(qb-q0) + mu*(Zy(x,y)-q0) + EmR*rq;
                    double dp=((lpp+sqr(rq))*A - rp*rq*B) / D;
                    double dq=((lpp+sqr(rp))*B - rp*rq*A) / D;


                    p2[x][y] = p0 + dp;
                    q2[x][y] = q0 + dq;

                    //bright_e+=sqr((e[x][y]-R(p[x][y],q[x][y]))/E0);
                    //grad_e+=sqr((e[x][y]-R(Zx(x,y),Zy(x,y)))/E0);
                    bright_e+=sqr(EmR/E0);
                    //grad_e+=sqr((get_e_prsp(x,y)-R(Zx(x,y),Zy(x,y)))/E0);
                    grad_e+=sqr(((1-
lrbal)*get_e1_prsp(x,y)+lrbal*get_e2_prsp(x,y)-R(Zx(x,y),Zy(x,y)))/E0);

                } //y
            } //x

            tempf=p; p=p2; p2=tempf;
            tempf=q; q=q2; q2=tempf;

            //warp z into stereo alignment?
            double warp=0;
            int px=0,py=0;
            if (i%pi==0) {
                pic++;
                px=pic%10*10 + (pic/100)%10;
                py=(pic/10)%10*10 + (pic/100)%10;

                double ste=getStereoError(px,py,4,0.);
                //if (ste>1/lambda)
                if (ste>.5/stereo_e)
                    warp=getBestWarp(px,py) * stereo_e / 40.0;
                //warp=getBestWarp(rx,ry)/4.0;
                System.out.println("Warp: ("+px+","+py+") "+warp+"  [error was
"+ste+" ]");
            }
            for(int x=0;x<=M;x++){
                for(int y=0;y<=N;y++){
                    // warp locally
                    double warpval=0.;
                    if (warp!=0) {
                        warpval = warp* Math.exp(-(sqr(px-x)+sqr(py-y)) /
(30*stereo_e * lambda));
                    }
                    z2[x][y] = warpval + Zbar(x,y) - sqr(SPACING)/K *
(Px(x,y)+Qy(x,y));

                    //calculate error from correct answer
                    err[x][y] = z2[x][y] + terrain.getSurfaceHeight(x-50,y-50);
                    abs_e+= sqr(err[x][y]);

                }
            }

            abs_e/=10000.0;
            alt_e=err2;
            tempz=z; z=z2; z2=tempz;
             if (i%di==0) {
                for(int x=0;x<M;x++){
                    for(int y=0;y<N;y++){
                        int_e+= sqr(Zx(x,y)-p[x][y])+sqr(Zy(x,y)-q[x][y]);
                    }
                }

                System.out.println(msg="i: "+i+" Br.e: "+bright_e+" Gr.e:
"+grad_e+" St.e: "+stereo_e+" int.e: "+int_e+" Abs.e: "+abs_e+" Alt.e:
"+alt_e+" lrbal: "+lrbal+" lambda:"+lambda+" mu: "+mu+" ;\r");
                msgs.insertElementAt(msg,0);

                if (grad_e<lambda*5) {
                    lambda=lambda*.666;
                    //mu=mu*.666;
                }

                // take progress shots at 0,100,400,900,1600,2500...
                if (Math.sqrt((double)i/100.)%1==0 && i%100==0){
                //int yoff=(int)Math.sqrt((double)i)*10;
                pc++;
                Graphics g=progress.getGraphics();
                int yoff=(pc-1)*102;
                g.drawImage(bi.getSubimage(330,0,430,100),0,yoff,null);
                g.setColor(Color.green);
                g.setFont(new Font("Arial",Font.PLAIN,9));
                g.drawString("i:"+i,113,yoff+10);
                g.setColor(Color.black);
                g.setFont(new Font("Courier",Font.PLAIN,11));
                g.drawString("i:"+i,432,yoff+12);
                g.drawString("br.e:"+bright_e,432,yoff+24);
                g.drawString("gr.e:"+grad_e,432,yoff+36);
                g.drawString("st.e:"+stereo_e,432,yoff+48);
                g.drawString("lambda:"+lambda,432,yoff+60);
                g.drawString("mu:"+mu,432,yoff+72);
                }
                repaint();
             }
```

```
    } //for i

}

public void dumpZ(int interval){
    System.out.println("----Begin Z dump------");
    for(int y=N;y>=0;y-=interval){
        for(int x=0;x<=M;x+=interval){
            System.out.print(z[x][y]+" ");
        }
        System.out.println();
    }
    System.out.println("----End Z dump------");
}


int camera_z=100;
int camera_b=100;

void init() {

    int screenw=400;//SCREEN_W;
    int screenh=200;//SCREEN_H;

    //terrain = new Terrain();
    terrain = new FractalTerrain(8,0.55,seed);
    contours = terrain.getSurfaceContours();

    System.out.println(0);

    Camera camera1 = new Camera(camera_b/2,0,camera_z,terrain);
    e1=camera1.getPerspectiveView2();

    System.out.println(1);

    Camera camera2 = new Camera(-camera_b/2,0,camera_z,terrain);
    e2=camera2.getPerspectiveView2();

    System.out.println(2);

    terrain.useR2=true;
    //e1b=camera1.getPerspectiveView2();
    try {
        Camera camera3 = new Camera(0,0,30000,terrain);
        e2b=camera3.getPerspectiveView();
    } catch (Exception e) { System.out.println(e); }
    terrain.useR2=false;

    repaint();
}

public void run() {
    this.init();
    //this.stereomatch();
    this.iterate();
    dumpZ(1);

}

public void stop() {
    thread.stop();
}
```

//

```
///////////////////////////////////////////////////////////////////////////////

    double[][][] SnTable;
    boolean[][][] SnOK;

    public void S_nReset(){
        SnOK=new boolean[100][100][DEPTH_RANGE];
    }

    public double S_n(int x, int y, int d){
        double s=0;
        try {
            if (SnOK[x][y][d]) return SnTable[x][y][d];
        } catch (Exception e) {
            return 0.0;
        }

        for (int xp=x-LS_BOUND;xp<=x+LS_BOUND;xp++){
            for (int yp=y-LS_BOUND;yp<=y+LS_BOUND;yp++){
                for (int dp=d-LS_BOUND;dp<=d+LS_BOUND;dp++){
                    try {
                        s+=L_n[xp][yp][dp];
                    } catch (Exception e) {}
                }
            }
        }
        SnTable[x][y][d]=s;
        SnOK[x][y][d]=true;
        return s;

    }

    double[][][] L_0, L_n, L_n1;
    int LS_BOUND=1; //3x3 (+/- 1)
    //int LS_BOUND=2; //5x5 (+/- 2)
    int[][] disparity = new int[100][100];
    int DEPTH_RANGE = 30;

    public void stereomatch(){
        L_0 = new double[100][100][DEPTH_RANGE];
        L_n = new double[100][100][DEPTH_RANGE];
        L_n1 = new double [100][100][DEPTH_RANGE];
        SnTable =new double[100][100][DEPTH_RANGE];

        // do initial match values
        for (int x=0;x<100;x++){
            for (int y=0;y<100;y++){
                for (int d=0;d<DEPTH_RANGE;d++){
                    // using squared differences between images
                    L_n[x][y][d] = L_0[x][y][d] =
                        1- Math.pow(e1[x+50][y+50]-e2[x+50+(d-
DEPTH_RANGE/2)][y+50],2.0);
                }
            }
        }


        // iterate
        for (int i=0;i<20;i++){
            S_nReset();
            System.out.println("sm i:"+i);
            for (int x=0;x<100;x++){
                for (int y=0;y<100;y++){

                    int dval=0;
                    double maxd=0;

                    for (int d=0;d<DEPTH_RANGE;d++){

                        double SumSn=-S_n(x,y,d);
                        for (int di=0;di<DEPTH_RANGE;di++){
                            SumSn+=S_n(x,y,di);
                            SumSn+=S_n(x+di-DEPTH_RANGE/2,y,di);
                        }
                        double a=2.0;
                        double R_n=Math.pow( S_n(x,y,d) / SumSn , a);

                        double dv= L_n1[x][y][d] = L_0[x][y][d] * R_n;
                        if (dv>maxd){
                            maxd=dv;
                            dval=d;
                        }

                    }
                    disparity[x][y]=dval;
                }
            }

            // dump L_n+1 into L_n  (actually swap them)
            double[][][] temp = L_n;
            L_n=L_n1;
            L_n1=temp;

            repaint();

        }
    }
}
```

```
/*
 * Camera.java
 *
 * Created on April 11, 2002, 4:05 PM
 */

/**
 *
 * @author  kwalker
 * @version
 */
public class Camera {

    Point3D position;
    Terrain terrain;
    double image[][];
    static int IMAGE_SIZE=200;
    static double PIXEL_SIZE=1;
    static double ACCURACY=0.5;

    /** Creates new Camera */
    public Camera(double _x,double _y, double _z, Terrain _terrain) {
          position = new Point3D(_x,_y,_z);
          terrain=_terrain;
    }
    // "Rel" functions convert terrain info into camera coordinates;
    double getSurfaceHeightRel(double _x, double _y)
    {
      _x+=position.x;
      _y+=position.y;
      return position.z-terrain.getSurfaceHeight(_x,_y);
    }

    double getSurfaceColorRel(double _x, double _y)
    {
      _x+=position.x;
      _y+=position.y;
      return terrain.getSurfaceColor(_x,_y);
    }

    double[][] getPerspectiveView()
    {

      image=new double [IMAGE_SIZE][IMAGE_SIZE];

      int sx=-IMAGE_SIZE/2-(int)Math.round(position.x/PIXEL_SIZE);
      int sy=-IMAGE_SIZE/2;
      int ex= IMAGE_SIZE/2-(int)Math.round(position.x/PIXEL_SIZE);
      int ey= IMAGE_SIZE/2;

      for (int y=sy; y<ey; y++){
          int ty=(int)(y*PIXEL_SIZE);
          for (int x=sx; x<ex; x++){
              int tx=(int)(x*PIXEL_SIZE);

              Point3D ray,ray_u;
              ray = new Point3D(tx,ty,position.z);
              ray_u = new Point3D(tx,ty,position.z);

              // 'normalize' to make the z component 1;
              ray_u.x /= ray_u.z;
              ray_u.y /= ray_u.z;
              ray_u.z /= ray_u.z;
              double height = getSurfaceHeightRel(ray.x,ray.y);

              while (Math.abs(ray.z-height)>ACCURACY){
                    // scale ray to have z=height
                    ray.x = ray_u.x * height;
                    ray.y = ray_u.y * height;
                    ray.z = ray_u.z * height;
                    height = getSurfaceHeightRel(ray.x,ray.y);
              }

              double color = getSurfaceColorRel(ray.x,ray.y);
              try {

image[x+IMAGE_SIZE/2+(int)Math.round(position.x*PIXEL_SIZE)][y+IMAGE_SIZE/2]=co
lor;
              } catch (Exception e) {
                  e.printStackTrace();

System.out.println(x+IMAGE_SIZE/2+(int)Math.round(position.x*PIXEL_SIZE)+","+y+
IMAGE_SIZE/2);
                  throw new RuntimeException();
              }
          }
      }

      return image;
    }

     double[][] getPerspectiveView2()
    {

      image=new double [IMAGE_SIZE][IMAGE_SIZE];

      int sx=-IMAGE_SIZE/2-(int)Math.round(position.x/PIXEL_SIZE);
      int sy=-IMAGE_SIZE/2;
      int ex= IMAGE_SIZE/2-(int)Math.round(position.x/PIXEL_SIZE);
      int ey= IMAGE_SIZE/2;

      for (int y=sy; y<ey; y++){
          int ty=(int)(y*PIXEL_SIZE);
          for (int x=sx; x<ex; x++){
              int tx=(int)(x*PIXEL_SIZE);

              Point3D ray,ray_u;
              ray = new Point3D(tx,ty,position.z);
              ray_u = new Point3D(tx,ty,position.z);

              // 'normalize' to make the z component 1;
              ray_u.x /= ray_u.z;
              ray_u.y /= ray_u.z;
              ray_u.z /= ray_u.z;
              double height = getSurfaceHeightRel(ray.x,ray.y);
              double step=Math.abs(position.z)/2.0;
              double probe=position.z;
              //trace rays
              int i=0;
              while (Math.abs(ray.z-height)>ACCURACY){
                //System.out.println("ray.z="+ray.z+" probe="+probe+"
height="+height+" step="+step);
                if (++i>100) throw new RuntimeException();
                    if (ray.z<height){
                        probe+=step;
                        step/=2.0;
                    } else {
                        probe-=step;
                        step/=2.0;
                    }
                    ray.x = ray_u.x * probe;
                    ray.y = ray_u.y * probe;
                    ray.z = ray_u.z * probe;
                    height = getSurfaceHeightRel(ray.x,ray.y);
                }

                double color = getSurfaceColorRel(ray.x,ray.y);
                try {

image[x+IMAGE_SIZE/2+(int)Math.round(position.x*PIXEL_SIZE)][y+IMAGE_SIZE/2]=co
lor;
                } catch (Exception e) {
                    e.printStackTrace();

System.out.println(x+IMAGE_SIZE/2+(int)Math.round(position.x*PIXEL_SIZE)+","+y+
IMAGE_SIZE/2);
                    throw new RuntimeException();
                }
            }
        }

        return image;
    }


    double[][] getOrthogonalView()
    {
        image=new double [IMAGE_SIZE][IMAGE_SIZE];
        int sx=-IMAGE_SIZE/2;
        int sy=-IMAGE_SIZE/2;
        int ex= IMAGE_SIZE/2;
        int ey= IMAGE_SIZE/2;

        for (int y=sy; y<ey; y++){
            int ty=(int)(y*PIXEL_SIZE);
            for (int x=sx; x<ex; x++){
                int tx=(int)(x*PIXEL_SIZE);
                image[x+IMAGE_SIZE/2][IMAGE_SIZE-(y+IMAGE_SIZE/2)] =
getSurfaceColorRel(tx,ty);
            }
        }
        return image;
    }

}
```

```
/*
 * terrain.java
 *
 * Created on April 11, 2002, 2:53 PM
 */
import java.awt.geom.Point2D.Double;
/**
 *
 * @author  kwalker
 * @version
 */
public class Terrain {

    //Point3D lightSource;
    public boolean useR2=false;

    /** Creates new terrain */
    public Terrain() {
    }

    public double getSurfaceHeight(double x, double y){
        //the terrain function
        //double z= 3*
Math.cos(Math.sqrt(Math.pow(x/3,2)+Math.pow(y/3,2)))+x/2;
        //double z= 3* Math.cos(Math.sqrt(Math.pow(x/3,2)+Math.pow(y/3,2))) +
5 * Math.cos(x/15)*Math.sin(y/15);;
        double z= 3*
Math.cos(Math.sqrt(Math.pow(x/3,2)+Math.pow(y/3,2)))+50*Math.sqrt(Math.pow(x/10
0,2)+Math.pow(y/100,2));
        //double z=  1 * Math.cos(x)*Math.sin(y);
        return z;
    }

    public Point2D getSurfaceGradient(double x,double y){
        double d=0.01;
        return new Point2D((getSurfaceHeight(x-d,y)-
getSurfaceHeight(x+d,y))/d/2,
        (getSurfaceHeight(x,y-d)-getSurfaceHeight(x,y+d))/d/2) ;
    }

    public double getSurfaceColor(double x, double y) {
        return getSurfaceAlbedo(x,y) * getSurfaceReflectance(x,y);
    }

    public double getSurfaceAlbedo(double x, double y) {
        return 1.0;  //constant for now
    }

    // reflectance map function
    public double R(double p, double q) {
        double qs=1;
        double ps=.5;
        double num= 1 + ps*p + qs*q;
        if (num <= 0) return 0;

        double denom=Math.sqrt(1+p*p+q*q)*Math.sqrt(1+ps*ps+qs*qs);

        return num/denom;
    }

    // alternate reflectance map function
    public double R2(double p, double q) {
        double qs=.5;
        double ps=-1;
        double num= 1 + ps*p + qs*q;
        if (num <= 0) return 0;

        double denom=Math.sqrt(1+p*p+q*q)*Math.sqrt(1+ps*ps+qs*qs);

        return num/denom;
    }

    public double getSurfaceReflectance(double x, double y)
    {
        Point2D gradient = getSurfaceGradient(x,y);
        if (useR2) return R2(gradient.x,gradient.y);
        else return R(gradient.x,gradient.y);
    }

    public double[][] getSurfaceContours(){
        double[][] img=new double[100][100];
        for (int x=0;x<100;x++){
            for (int y=0;y<100;y++){
                double zval=1;
                double z=getSurfaceHeight(x-50,y-50);
                if (Math.abs(z)%1<.25) {
                    zval=0;
                }
                else if (z>0) zval=.75;
                img[x][y]=zval;
            }
        }
        return img;
    }

}
```

```
/*
 * FractalTerrain.java
 *
 * Created on April 30, 2002, 2:41 PM
 */
import java.util.*;
/**
 *
 * @author  kwalker
 * @version
 */

public class FractalTerrain extends Terrain {
  private double[][] terrain;
  private double roughness, min, max;
  private int divisions;
  private Random rng;

  public FractalTerrain (int lod, double roughness, long seed) {
    this.roughness = roughness;
    this.divisions = 1 << lod;
    terrain = new double[divisions + 1][divisions + 1];
    rng = new Random(seed);

    terrain[0][0] = rnd ();
    terrain[0][divisions] = rnd ();
    terrain[divisions][divisions] = rnd ();
    terrain[divisions][0] = rnd ();

    double rough = roughness ;
    for (int i = 0; i < lod; ++ i) {
      int q = 1 << i, r = 1 << (lod - i), s = r >> 1;
      for (int j = 0; j < divisions; j += r)
        for (int k = 0; k < divisions; k += r)
          diamond (j, k, r, rough);
      if (s > 0)
        for (int j = 0; j <= divisions; j += s)
          for (int k = (j + s) % r; k <= divisions; k += r)
            square (j - s, k - s, r, rough);
      rough *= roughness * .65;
    }

    min = max = terrain[0][0];
    for (int i = 0; i <= divisions; ++ i)
      for (int j = 0; j <= divisions; ++ j)
        if (terrain[i][j] < min) min = terrain[i][j];
        else if (terrain[i][j] > max) max = terrain[i][j];
  }

  private void diamond (int x, int y, int side, double scale) {
    if (side > 1) {
      int half = side / 2;
      double avg = (terrain[x][y] + terrain[x + side][y] +
        terrain[x + side][y + side] + terrain[x][y + side]) * 0.25;
      terrain[x + half][y + half] = avg + rnd () * scale;
    }
  }

  private void square (int x, int y, int side, double scale) {
    int half = side / 2;
    double avg = 0.0, sum = 0.0;
    if (x >= 0)
    { avg += terrain[x][y + half]; sum += 1.0; }
    if (y >= 0)
    { avg += terrain[x + half][y]; sum += 1.0; }
    if (x + side <= divisions)
    { avg += terrain[x + side][y + half]; sum += 1.0; }
    if (y + side <= divisions)
    { avg += terrain[x + half][y + side]; sum += 1.0; }
    terrain[x + half][y + half] = avg / sum + rnd () * scale;
  }

  private double rnd () {
    return 2. * rng.nextDouble () - 1.0;
  }

  public double getAltitude (double i, double j) {
    try {
    int x1=(int)Math.floor(i * divisions);
    int x2=(int)Math.ceil(i * divisions);
    int y1=(int)Math.floor(j * divisions);
    int y2=(int)Math.ceil(j * divisions);
    double x1d=(i*divisions)-x1;
    double x2d=x2-(i*divisions);
    double y1d=(j*divisions)-y1;
    double y2d=y2-(j*divisions);
    double d11=Math.pow(x1d*x1d+y1d*y1d,.5);
    double d12=Math.pow(x1d*x1d+y2d*y2d,.5);
    double d21=Math.pow(x2d*x2d+y1d*y1d,.5);
    double d22=Math.pow(x2d*x2d+y2d*y2d,.5);

    double alt= ((1-d11)*terrain[x1][y1]+
                 (1-d12)*terrain[x1][y2]+
                 (1-d21)*terrain[x2][y1]+
                 (1-d22)*terrain[x2][y2])/(4-d11-d12-d21-d22);

    //double alt = terrain[(int) (i * divisions)][(int) (j * divisions)];
    return (alt - min) / (max - min);
    } catch (ArrayIndexOutOfBoundsException e) {
        return 0.0;
    }
  }

  /*
  public double getAlt(double i,double j){
    return getAltitude((i+128)/256.0,(j+128)/256.0) * 10.0 - 5;
  }
  */

  public double getSurfaceHeight(double i,double j){
      return getAltitude((i+128)/256.0,(j+128)/256.0) * 40.0 - 20;


  }
  public Point2D getSurfaceGradient(double x,double y){
      try {
        double d=0.5;
        return new Point2D((getSurfaceHeight(x-d,y)-
getSurfaceHeight(x+d,y))/d/2,
        (getSurfaceHeight(x,y-d)-getSurfaceHeight(x,y+d))/d/2) ;
      } catch (Exception e) {
          e.printStackTrace();
          throw new RuntimeException();
      }
    }
}
```