The Floating Column Algorithm for Shaded, Parallel Display of Function Surfaces without Patches

Dan Gordon

Abstract—The *floating column algorithm* is a new method for the shaded rendering of function surfaces. Derived from the monochromatic floating horizon algorithm, it uses the partial derivatives of the function to compute surface normals, thus enabling intensity or normal-interpolation shading. Current rendering methods require tiling the surface with patches, so higher resolution patching is required for zoom-in views or interactive modification or time-varying surfaces. The new algorithm requires no patching and uses only constant space, so it can be implemented on graphics cards and hand-held devices. Each pixel-column is displayed independently of the others, and this "independent column mode" makes the algorithm inherently parallel in image-space, so it is suitable for multiprocessor workstations and clusters and it is scalable in the resolution size. Furthermore, the sampling frequency of the surface can be controlled locally, matching local surface features, distance, or artifact elimination requirements. Space-efficient supersampling for antialiasing is also possible. The new algorithm, which allows orthogonal and perspective projections, produces pixel-wide strips which can be displayed in software or hardware. Various extensions are described, including shadows and texture mapping. These properties, together with the algorithm's parallelism, make it potentially useful for the real-time display of functionally defined textured terrains and the animated display of time-varying surfaces.

Index Terms—Floating column, floating horizon, function display, functionally defined terrains, height-fields, graphics hardware, local supersampling, mathematical software packages, parallel rendering.

1 Introduction and Motivation

The floating horizon algorithm is an old and well-known technique for displaying the graph of a function of two variables—see [15], [16], [12], [11], [9], [7]. Originally designed for plotters and vector displays, its output consists of monochromatic lines. Although the output is not shaded, the proximity of the lines provides a visual cue to the curvature of the resulting surface. The main advantage of this algorithm is its speed and low memory requirement—just $O(m)$ memory locations are required, where m is the number of vertical samples chosen for a particular display. For raster displays, m is usually just the number of pixels per scanline.

The low price and widespread availability of high-quality color displays call for better quality visualization techniques. The standard method for producing a shaded display is to fit the surface with polygonal patches—usually triangles or quadrilaterals—see, for example, Glaeser [5, Section 5.4]. The patches are then displayed by standard display methods, which are also available in hardware on today's workstations. One advantage of this approach is that different views of the same function do not need repatching the surface. However, patch-fitting has some drawbacks:

- Zooming into a small area causes the patches to become noticeable. This problem can only be resolved by viewing the surface with a higher resolution, which means repatching the surface or using multiresolution techniques.
- Zooming out creates an opposite problem: Time is wasted on rendering patches that are very small compared to the requirement set by the display device. Multiresolution, if available, can solve this problem, but at the cost of the extra space required for multiresolution.
- Modifying the function during interactive viewing, or viewing a time-varying surface, requires a repatching of the surface.
- The required memory is proportional to the number of patches and this depends on the desired approximation.

The last-mentioned problem is usually not serious for viewing a single mathematical function, but it could be problematic in an application such as virtual flying over a functionally defined terrain, in games or flight-simulators. By "functionally defined terrain" we mean that there is some method or function which provides the height at any given point.

In this paper, we present a new approach to the problem of rendering function surfaces. The new method, called the *floating column algorithm*, draws on the floating horizon algorithm with some important modifications. As in the floating horizon, visualization proceeds from front to back and the minimum and maximum set pixels are maintained. In addition, some other value(s), such as intensities or

• *The author is with the Department of Computer Science, University of Haifa, Haifa 31905, Israel. E-mail: gordon@cs.haifa.ac.il.*

Manuscript received 3 Sept. 1999; revised 13 Sept. 2000; accepted 2 Mar. 2001.
For information on obtaining reprints of this article, please send e-mail to: tvcg@computer.org, and reference IEEECS Log Number 110541.

surface normals are maintained and used for surface shading. The surface normals are evaluated by using the partial derivatives of the function. The algorithm produces pixel-wide strips which can be displayed using software or graphics hardware. No surface patching is required, but one may view the creation of the pixel-wide strips as a form of "on-the-fly" patching, done at the exact required image-space resolution and applied only to the visible parts.

The partial derivatives of the function can either be supplied by the user or they can be approximated numerically [10, chapter 4]. The algorithm itself (without the function or the image memory) requires only constant space, so it can fit in ROM on a graphics card and be executed by the card's processor(s). It is also suitable for limited-memory environments, such as hand-held devices. The floating column algorithm operates in *independent column mode*, by which we mean that every column is rendered independently of the others (hence the name of the new method). This characteristic, not shared by the regular floating horizon, provides the new method with several important properties:

1. The floating column algorithm is inherently parallel in image-space, so it is ideal for multiprocessor workstations which are becoming increasingly commonplace. Other parallel machines and clusters can also utilize this property. If implemented on a graphics card, several of the card's processors can run it in parallel.
2. From a parallel programming viewpoint, the algorithm is scalable in the resolution, meaning that resolution can be increased indefinitely by a proportional increase in the number of processors. This makes the algorithm ideal for current and future huge—even wall-sized—displays. Applications of this property in virtual reality and flight simulators are obvious.
3. Supersampling can be controlled locally since it can be restricted both column-wise and step-wise, where a step is the distance between the successive "horizons" that sample the surface. Thus, for a function, the sampling frequency can be matched to its local oscillation frequency. For functionally defined terrains, sampling can depend on local surface features or distance. The algorithm thus easily provides a simple level-of-detail adaptation.
4. The column mode is also useful for antialiasing: Columns can be sampled at a higher frequency and averaged, with only two to three image columns required to be in memory at any time.

With perspective projections, very large areas of the function surface can be viewed and the function can be modified interactively without repatching—this is useful for function visualization in mathematical software packages or for viewing functionally defined terrains in virtual reality games or flight simulators. It is also potentially useful for the animated display of liquid surfaces or, in general, time-varying surfaces. Another feature is the ability to color the two sides of a function surface with different colors, thus enhancing its visualization. Various extensions of the basic algorithm are detailed later.

The floating column produces some artifacts when the sampling distance is not very small—see Figs. 8 and 13. These are corrected by a variety of methods, such as local supersampling, spline-fitting, and normal interpolation shading. Another problem is that discontinuities in the function surface produce vertical "walls" which are not sampled, so they are not shaded correctly; suggestions for tackling this problem are discussed in Section 5.

The rest of the paper is organized as follows: Section 2 places the new method in relation to previous work. Sections 3 and 4 present the geometric details and the basic floating column algorithm. Section 5 details the artifact correction methods, while Section 6 outlines several extensions of the new method. Sections 7 and 8 conclude with some results and a discussion.

2 Applicability and Relation to Previous Work

The earliest published references on the floating column algorithm are apparently due to Williamson [15], Wright [16], and Watkins [12]. See also Rogers [11, Section 4.2] and Pokorny and Gerald [9, Section 13.1], with the latter also presenting the cross-hatched version of the algorithm. A relatively recent improvement is due to Kohl [7], whose method corrects problems with previous approaches and enables perspective projections and arbitrary viewing orientations. A natural application of the floating horizon is the display of functions in mathematical software packages.

One question that arises is whether the floating column is useful for surfaces represented by a mesh of patches. The answer to that lies in the precise form of surface representation. For most applications, there are two different methods of representing curved surfaces:

A. The surface is represented by a mesh of flat polygons —usually triangles and/or quadrilaterals and the same surface is represented by several levels of resolution. This topic has received a lot of attention in recent years, following Hoppe [6] and others. The polygons can be directly displayed using current graphics hardware and the floating column method is *not* helpful when multiresolution representation is available.
B. The surface is represented by only one resolution of patches, which may be flat or curved. In this case, every point in the surface domain has an associated height which can be used by the floating column. If the vertices of the patches form a regular grid in the surface domain, then we can easily find the height for each point. If not, then we may need some geometric data structure for this purpose. Such a structure, based on bucketing techniques, can consist of a regular grid of squares, with each square associated with a list of all surface patches whose projection on the surface domain intersects that square. To find the height associated with a point in the plane, we first determine the square containing

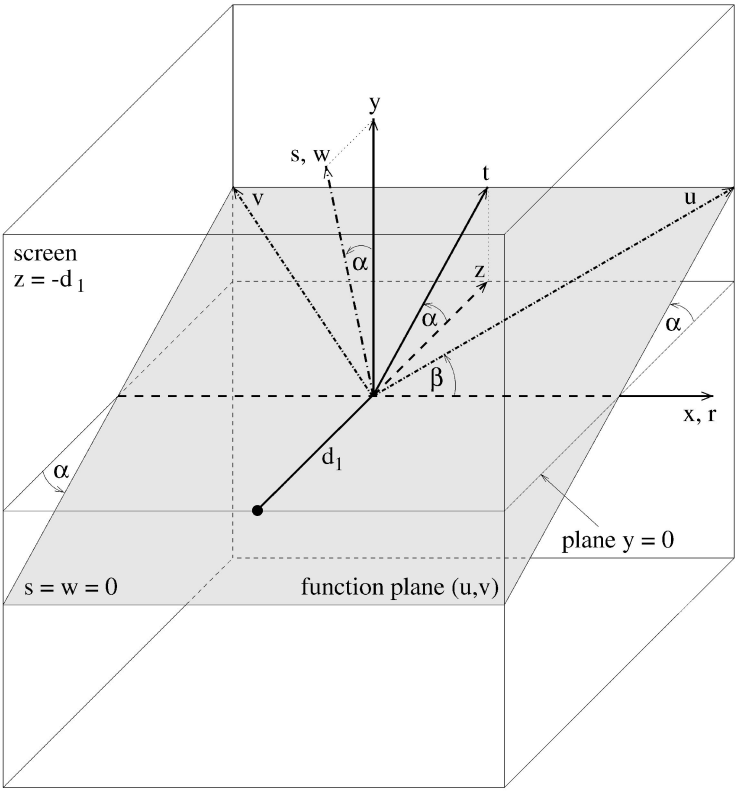

Fig. 1. The two rotations used for viewing a function.

the point and then search the square's list for the appropriate patch.

The above remarks apply to function surfaces and to terrains. Thus, for the case of terrains, the applicability of the floating column depends on whether the terrain is represented in form **A** or **B** as described above. The floating column is naturally applicable when the terrain is described as a function, in games and flight simulators. Throughout the rest of the paper, whenever reference is made to terrains, the assumption is that the terrain is represented in a form that can be displayed by the floating column.

A widely used form for surfaces is the *digital height field*, where the surface is represented by its height over the set of grid points of some rectangular coordinate system in the plane. Clearly, the floating column can be used for functions represented in this form (for any (x, y), the height is a linear interpolation of the adjacent grid points and the partial derivatives can be approximated numerically). Recently, Albersmann et al. [1] presented a technique for the shaded display of digital height fields without the use of polygonal patches. Their method is restricted to digital height fields and it lacks several of the properties of the floating column, such as perspective projections and local supersampling.

One of the modern approaches to representing terrains is through the use of *voxels*. Cohen-Or et al. [2] use an efficient form of ray casting and parallel processing to obtain real-time rendering of terrains represented by voxels. Voxels are also used for realistic terrain representation in games—see [8]. If the texture is just an aerial photo, then the voxel terrain is simply a digital height field with a texture, and this can be handled by the new algorithm (Section 6 explains texture mapping and other extensions). However, if surface features are present in the voxel model, they may be missed by regularly spaced samples of the horizons. This can be rectified either by closely spaced samples or by some mechanism that will induce supersampling in the area of the surface features. However, if the features contain vertical components, they are best handled with a hardware Z-buffer—see Section 6.1

3 Geometric Preliminaries

This section presents the geometric basics for our modified floating horizon and the floating column algorithm.

3.1 Basic Transformations

We use three coordinate systems, as shown in Fig. 1: x, y, z are the screen coordinates and r, s, t are obtained from the screen coordinates by tilting the $y = 0$ plane by an angle α about the *x*-axis, which is also the *r*-axis. The resulting plane, whose equation is $s = 0$, is the one on which the function $w = f(u, v)$ is defined. The u, w, v coordinate system is obtained from the r, s, t system by rotating the r, t axes by an angle β about the *s*-axis, which is also the *w*-axis. The relations between the three coordinate systems are given by (1) and (2):

$$\begin{pmatrix} r \\ s \\ t \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\alpha & \sin\alpha \\ 0 & -\sin\alpha & \cos\alpha \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} \tag{1}$$

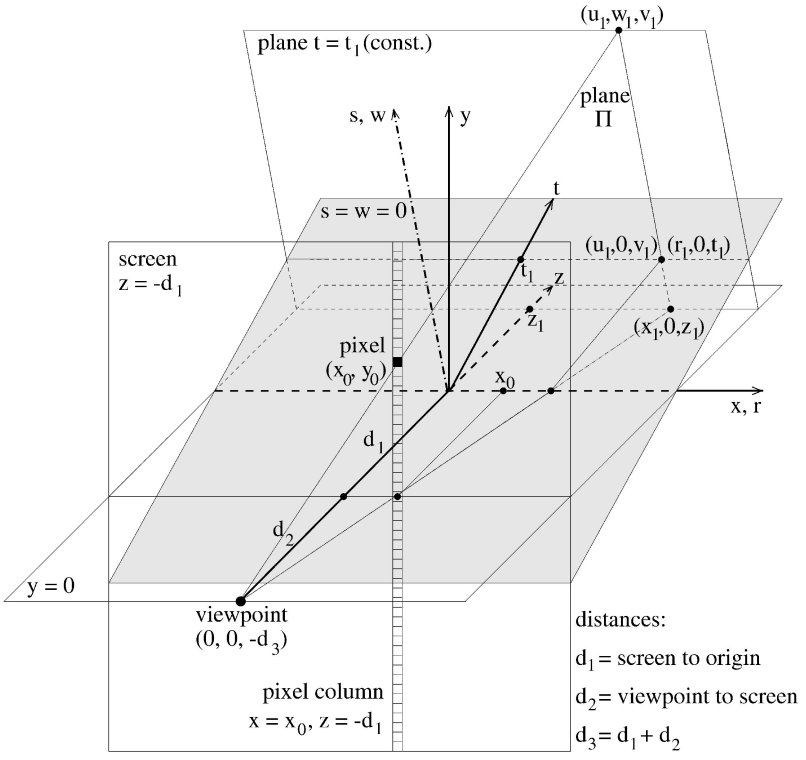

Fig. 2. The geometry of the perspective view.

$$\begin{pmatrix} u \\ w \\ v \end{pmatrix} = \begin{pmatrix} \cos\beta & 0 & \sin\beta \\ 0 & 1 & 0 \\ -\sin\beta & 0 & \cos\beta \end{pmatrix} \begin{pmatrix} r \\ s \\ t \end{pmatrix}. \tag{2}$$

Clearly, the above transformations are sufficient to view the surface of the function from any direction. If required, a rotation of the final image about the screen z-axis can be used to view the function with any "up" vector, but this is seldom needed for function visualization. Such a rotation is a standard image-space operation which is even implemented in hardware on modern equipment. Note that we have assumed, for simplicity, that the screen coordinate system is centered about the origin of the function and that it is scaled to be identical to the function system u, v, w. The actual device coordinates can be obtained from this by a scaling and translation transformation.

3.2 Orthographic Projections

For orthographic projections, the required geometric transformations are as follows: Given a column of pixels defined by the plane $x = x_0$, and a plane of constant t, $t = t_1$, we determine the intersection of these planes with the (u, v)-plane, (u_1, v_1) from (2) by:

$$\begin{pmatrix} u_1 \\ v_1 \end{pmatrix} = \begin{pmatrix} \cos\beta & \sin\beta \\ -\sin\beta & \cos\beta \end{pmatrix} \begin{pmatrix} x_0 \\ t_1 \end{pmatrix}. \tag{3}$$

Note that we have used x_0 in place of r since $r = x = x_0$ according to (1).

We now compute $w_1 = f(u_1, v_1)$, where f is the function we want to display. The point that we need to display is (u_1, w_1, v_1) and we denote this point by P. Let (x_0, y_0, z_0) be P's coordinates in image space. From (2), we know that the s-coordinate of P is simply w_1, so, from (1), we get:

$$\begin{pmatrix} y_0 \\ z_0 \end{pmatrix} = \begin{pmatrix} \cos\alpha & -\sin\alpha \\ \sin\alpha & \cos\alpha \end{pmatrix} \begin{pmatrix} w_1 \\ t_1 \end{pmatrix}. \tag{4}$$

3.3 Perspective Projections

The standard procedure for perspective projections in computer graphics starts from the point to be displayed, which is known, and applies all the necessary transformations, including the perspective transformation—see [4, chapter 6]. Our method of obtaining a perspective projection of a function is different because we do not start from some given set of points on the surface. We start from a given viewpoint and a column of pixels, choose a sampling plane, and then obtain a point to be displayed. Fig. 2 shows the geometry of the perspective method.

Let d_1 be the distance of the screen from the origin and d_2 the distance of the viewpoint from the screen, as shown in Fig. 2. We also denote $d_3 = d_1 + d_2$, so the coordinates of the viewpoint (in image space) are $(0, 0, -d_3)$. The viewpoint is on the z-axis for simplicity. A column of pixels can be regarded as a straight line defined by the two equations $x = x_0$ and $z = -d_3$. The pixel column and the viewpoint determine a plane Π, which contains all the surface points which project onto the given pixel column.

The plane Π, together with a given sampling plane $t = t_1$, where t_1 is a constant, determines a unique point $P = (u_1, w_1, v_1)$ on the function surface—this is the point that we need to display. To find P, we first find the point $(r_1, 0, t_1)$, which is the intersection of the three planes Π, $t = t_1$, and the plane $s = 0$ (which is identical to $w = 0$). After finding r_1 (t_1 is given), we find u_1 and v_1 by using (2). We now

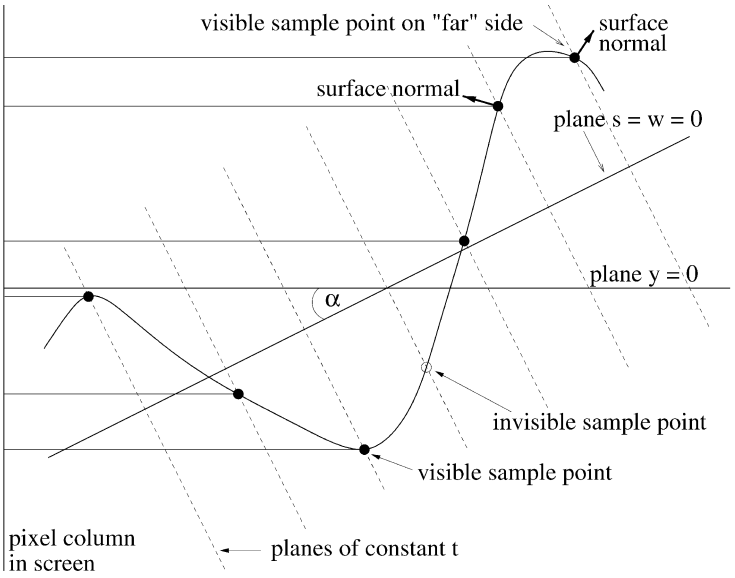

Fig. 3. The basic hidden surface technique.

compute $w_1 = f(u_1, v_1)$ to get P. The line segment connecting this point to the viewpoint determines the pixel (x_0, y_0) to be set.

Determining r_1 is a somewhat tedious calculation, so we only present the main steps and their results. Let $Ar + Bs + Ct + D = 0$ be the equation of Π in the r, s, t system. Clearly, Π intersects the r-axis, so $A \neq 0$ and we can assume $A = 1$. The point $(r_1, 0, t_1)$ is in Π, so $r_1 + Ct_1 + D = 0$, from which we get

$$r_1 = -Ct_1 - D. \tag{5}$$

We derive three equations which determine the values of B, C, and D as follows: The viewpoint is in Π and, from (1), its r, s, t coordinates are $(0, -d_3 \sin\alpha, -d_3 \cos\alpha)$. So, we get:

$$B(-d_3 \sin\alpha) + C(-d_3 \cos\alpha) + D = 0. \tag{6}$$

The pixel column is the set $\{(x_0, y, -d_2) \mid y \in \mathbb{R}\}$, where $\mathbb{R}$ is the set of real numbers. From (1), the r, s, t coordinates of a point $(x_0, y, -d_2)$ in the pixel column are $(x_0, y\cos\alpha - d_2 \sin\alpha, -y\sin\alpha - d_2 \cos\alpha)$. Every such point satisfies Π's plane equation, i.e., for every $y \in \mathbb{R}$, we have

$$x_0 + B(y\cos\alpha - d_2 \sin\alpha) + C(-y\sin\alpha - d_2 \cos\alpha) + D = 0. \tag{7}$$

By taking $y = 0$ and $y = 1$ in (7), we get two more equations, which, together with (6), produce:

$$A = 1, B = C\tan\alpha, C = D\cos\alpha/d_3, \text{and } D = -x_0 d_3/d_1. \tag{8}$$

These identities, together with (5) result in

$$r_1 = x_0(t_1 \cos\alpha + d_3)/d_1. \tag{9}$$

From r_1 and t_1, we obtain $P = (u_1, w_1, v_1)$ as described above. To display P, we need its coordinates (x_1, y_1, z_1) in the x, y, z system. Since w_1 is P's s-coordinate and t_1 is its t-coordinate, we get, from (1):

$$\begin{pmatrix} y_1 \\ z_1 \end{pmatrix} = \begin{pmatrix} \cos\alpha & -\sin\alpha \\ \sin\alpha & \cos\alpha \end{pmatrix} \begin{pmatrix} w_1 \\ t_1 \end{pmatrix}. \tag{10}$$

Using the standard similarity relations, we have $x_1 = x_0(z_1 + d_3)/d_1$, which completes the evaluation of (x_1, y_1, z_1). Projecting this point toward the viewpoint, we get the screen point $(x_0, y_0, -d_1)$, where $y_0 = y_1 d_1/(z_1 + d_3)$.

4 The Floating Column Algorithm

4.1 A Modified Floating Horizon Algorithm

We modify the regular algorithm in order to adapt it later for shading. The modified algorithm operates as follows: The function plane $s = w = 0$ is sampled at regular intervals by a perpendicular plane of the form $t = \text{const.}$—see Figs. 2 and 3. For every pixel column, we maintain the positions of the minimum and maximum displayed pixel. As the sampling plane advances, the projection of each surface point is compared against the current minimum and maximum and displayed only if it is visible.

The algorithm can operate in independent column mode, but the result would not be acceptable because steep lines would be sampled only once per column and this would cause them to appear disconnected. We therefore proceed as in the regular floating horizon algorithm, i.e., the "horizon" proceeds from front to back and, at each column, each new point is connected to the last visible point in the previous column. Even if the new point is not visible, a partial line may still be needed to prevent "dangling" lines. For these purposes, we maintain, for each column, the index of the last-drawn pixel (which is either the minimum or the maximum).

The above algorithm differs from the usual floating horizon, in which the sampling planes are parallel to the u- or v-axis of the function domain, while a plane of constant t is parallel to the screen x-axis. The sampled columns do not need to be adjacent. The algorithm proceeds by an outer loop on the sampling planes, with an inner loop on the columns. The required memory is $\Theta(m)$, where m is the

number of sampled pixel columns. The detailed algorithm is presented in Appendix A, Section A.1.

4.2 Overview of the Floating Column Algorithm

There are two main changes from the regular floating horizon algorithm: The independent column mode and the maintenance of the value(s) to be interpolated. Since each pixel column is handled independently of the others, the columns can be filled in any sequential order or in parallel. Only $O(1)$ memory is required for a single column, so, in a sequential implementation, this is all of the required memory. Of course, this does not include any memory that may be required by the viewed function or for the screen.

When rendering a single column, the $t = \text{const.}$ plane is advanced in front to back order as in the floating horizon. In addition to the minimum, maximum, and previous pixels of the column, we also maintain the last value (or values) that we wish to interpolate. At each new visible point, all the pixels from the previous extremum to the current one are filled by interpolating from the previous value(s) to the new ones. The value(s) are computed and stored at each point, even if the surface is not visible at that point, since they are required for interpolation in case the next point is visible.

An example can be seen in Fig. 3: The fourth sample point is invisible, but the values that need to be interpolated are computed and stored for this point. The fifth point is visible and the values at the fourth point are used for interpolation in order to fill all the pixels between the previous maximum and the current maximum. Note that the last point, on the "far" side of a "ridge," is considered as visible in the simplistic implementation since its y-value is greater than the current maximum. That's one reason for the artifacts, which are eliminated by supersampling the in-between region or by fitting a cubic spline to find the top of the ridge.

We can distinguish between the two sides of the surface and shade them with different colors: If the new point is above the maximum, the viewed point is on the upper surface; otherwise, it is on the lower surface. For the very first point in a pixel column, we use the sign of the dot product between the surface normal and the view vector in order to distinguish between the two cases.

The surface normal is computed by using the partial derivatives of the function. These can either be supplied by the user or they can be computed numerically if only the function is supplied; the former is always preferable since it is more accurate. The surface normal can be used for intensity interpolation shading ("Gouraud" shading); in this case, the stored value is just the computed intensity, calculated according to any desired model, such as diffuse reflection.

Instead of intensity interpolation, we can perform normal interpolation and this requires the storage of the three coordinates of the surface normal at each sample point. Normal interpolation has two advantages over intensity interpolation: The shading artifacts are greatly diminished and specular reflection ("Phong" shading) can also be performed. Specular reflection has not been implemented in the current version of the program.

4.3 Detailed Calculations

The computation of each point to be displayed is done as explained in Section 3. Additional calculations for producing a shaded display are done as follows: Denote by $\mathbf{N}$ the vector normal to the surface. $\mathbf{N}$'s coordinates in the function coordinate system u, v, w are given by $(N_u, N_w, N_v) = (-f'_u, 1, -f'_v)$, where $f'_u = \partial f/\partial u$ and $f'_v = \partial f/\partial v$. From (1) and (2), $\mathbf{N}$'s screen coordinates are given by

$$\begin{pmatrix} N_x \\ N_y \\ N_z \end{pmatrix} = \begin{pmatrix} \cos\beta & 0 & -\sin\beta \\ -\sin\alpha\sin\beta & \cos\alpha & -\sin\alpha\cos\beta \\ \cos\alpha\sin\beta & \sin\alpha & \cos\alpha\cos\beta \end{pmatrix} \begin{pmatrix} -f'_u \\ 1 \\ -f'_v \end{pmatrix}. \tag{11}$$

We assume, for simplicity, that the light source and the viewpoint coincide. Other light sources can be easily implemented, with the exception of sources which cast shadows. The issue of shadows will be discussed later, in Section 6. We denote by $\mathbf{L}$ the vector from the displayed point toward the light source. The displayed point is (x_1, y_1, z_1) and the coordinates of the viewpoint are $(0, 0, -d_3)$. Hence, $\mathbf{L}$'s screen coordinates are $(-x_1, -y_1, -(d_3 + z_1))$. From this and from (11), we can compute the dot product of $\mathbf{L}$ and $\mathbf{N}$:

$$\begin{aligned} \mathbf{L}\cdot\mathbf{N} = {} & (x_1(f'_u\cos\beta - f'_v\sin\beta) \\ & - y_1(f'_u\sin\alpha\sin\beta + \cos\alpha + f'_v\sin\alpha\cos\beta) \\ & - (d_3 + z_1)(-f'_u\cos\alpha\sin\beta + \sin\alpha - f'_v\cos\alpha\cos\beta))/ \\ & (\|\mathbf{L}\|\|\mathbf{N}\|), \end{aligned} \tag{12}$$

where $\|\cdot\|$ denotes the length of a vector.

The floating column algorithm is presented in detail in Appendix A, Section A.2.

5 Artifact Elimination

5.1 Problem Description and Solution Methods

Three artifacts appear when the new method is implemented. These problems could be eliminated by sampling the function at a uniformly high frequency, but such an approach is inefficient. The most noticeable artifact, which can be seen in Fig. 8, is the appearance of triangular shading patches near ridges when intensity interpolation shading is used. The second problem, which is quite pronounced in the top left of Fig. 8, is the corrugated appearance of some ridges. This is due to the fact that the sampling planes form an oblique angle with the ridges. The corrugations also appear in the regular floating horizon, but this is acceptable for line drawings.

A third problem, shown in the bottom part of Fig. 13, occurs with intensity interpolation when a ridge or valley is viewed from above: If the ridge is not parallel to the sampling planes, then, at some of the pixel columns, it will be missed and, at some, it will be sampled exactly. When such a ridge is sampled near the top, the diffuse shading equation [4, p. 724] produces a bright value due to the small angle between the surface normal and the view vector. When the ridge is missed by the sampling planes, intensity

interpolation shading will shade the top by some value between the two values at the sampled sides of the ridge, which are darker. For convenience, we refer to this as the *bump problem*. Three different methods, well-known from the literature, were implemented to handle these problems.

Spline fitting: When a ridge is detected, its top is fitted with a cubic spline and the function is sampled at the top of the spline. The details are described in Section 5.2.

Local supersampling: Whenever a problem is detected between two sample points, the function is sampled at a higher frequency between these points. This method can eliminate all the above problems. Problem detection depends on what we want to correct: Ridge detection is described in the following section, while shading problems are corrected by ensuring that intensities are limited to some user-defined threshold value. The detection and correction of the bump problem are detailed in Section 5.3 below.

Normal interpolation shading: This method greatly reduces the triangular shading artifacts near ridges and it naturally avoids the bump problem.

Another problem is that discontinuities in the function appear as a vertical wall connecting two surfaces. The wall will not be shaded correctly by intensity interpolation since, no matter how closely the surface is sampled, the wall itself is not sampled. The correct way to handle this problem is first to detect it, then approximate the wall region with a tightly fitted but continuous function, and then use normal interpolation shading. The problem can be detected when two successive plane normals point "approximately" upward, but there is a "big" jump in the y-values. The terms in quotes mean that threshold values should be used. This topic has not been implemented in the current version of the program. Note that the modified floating horizon handles such cases correctly because no shading is involved.

5.2 Spline-Fitting

Consider the intersection of the function surface and the plane Π, as shown in Fig. 4. Currently, spline-fitting is implemented only for orthographic projections, but the detailed presentation can be modified for perspective projections. Suppose that, at two successive values of t, t_0 and t_1, we found that the dot products of the view vector and the normals to the surface are of opposite signs. If we sample the surface at t_0 and then at t_1, we will miss the top of the ridge and this produces a corrugated effect, as shown in Fig. 8. Our solution is to fit a cubic spline between the two sample points, find the extreme point of the spline, and then to consider that extreme point as if it were a regular sample point of the surface.

Assume that the spline is given by the cubic polynomial $s = p(t) = at^3 + bt^2 + ct + d$, so its derivative is $p'(t) = 3at^2 + 2bt + c$. Let s_0, s_1 be, respectively, the values of the function surface at t_0 and t_1. s_0, s_1 are simply w_0, w_1, obtained from t_0, t_1, as described in Section 2. Since the spline must pass through the points $(t_0, s_0), (t_1, s_1)$, we get two equations:

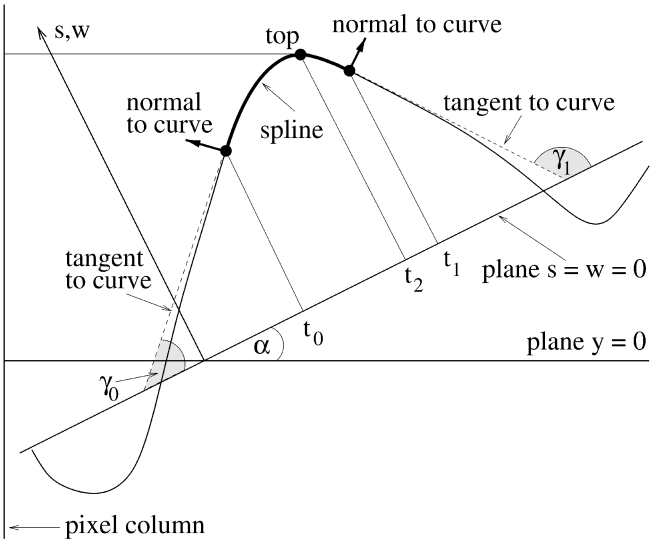

Fig. 4. Spline fitting between two surface points.

$$p(t_0) = at_0^3 + bt_0^2 + ct_0 + d = s_0 \tag{13}$$

$$p(t_1) = at_1^3 + bt_1^2 + ct_1 + d = s_1. \tag{14}$$

Furthermore, since we want the spline to have the same normal as the curve at the two points, we get two more equations:

$$p'(t_0) = 3at_0^2 + 2bt_0 + c = \tan\gamma_0 \tag{15}$$

$$p'(t_1) = 3at_1^2 + 2bt_1 + c = \tan\gamma_1, \tag{16}$$

where $\tan\gamma_0$ and $\tan\gamma_1$ are, respectively, the slopes of the curve at (t_0, s_0) and (t_1, s_1). Denoting by Δt the step size $t_1 - t_0$ and solving the four equations, we get:

$$\left.\begin{aligned} a &= \frac{2(s_0 - s_1)}{\Delta t^3} + \frac{\tan\gamma_0 + \tan\gamma_1}{\Delta t^2} \\ b &= \frac{\tan\gamma_1 - \tan\gamma_0}{2\Delta t} - \frac{3a(t_0 + t_1)}{2} \\ c &= \tan\gamma_1 - (3at_1 + 2b)t_1 \\ d &= s_1 - ((at_1 + b)t_1 + c)t_1 \end{aligned}\right\} \tag{17}$$

The slopes $\tan\gamma_i$, for $i = 0, 1$, are evaluated as follows: Let (u_i, v_i), $i = 0, 1$, be the u, v values corresponding to t_0, t_1, as computed in Section 3.2, and let $w_i = f(u_i, v_i)$. Denoting by $\mathbf{N}^i$, $i = 0, 1$, the normals to the surface at the points (u_i, w_i, v_i), we have, from Section 4.3:

$$(\mathbf{N}_u^i, \mathbf{N}_w^i, \mathbf{N}_v^i) = (-f_u'(u_i, v_i), 1, -f_v'(u_i, v_i)). \tag{18}$$

Let $(\mathbf{N}_r^i, \mathbf{N}_s^i, \mathbf{N}_t^i)$ be the components of $\mathbf{N}^i$ in the r, s, t coordinate system. Then, in the plane Π, the normals to the curve at the two points are the vectors $(\mathbf{N}_s^i, \mathbf{N}_t^i)$. From (2), we have

$$\mathbf{N}_s^i = \mathbf{N}_w^i, \quad \mathbf{N}_t^i = \mathbf{N}_u^i \sin\beta + \mathbf{N}_v^i \cos\beta, \tag{19}$$

from which we get:

$$\mathbf{N}_s^i = 1, \quad \mathbf{N}_t^i = -f'_u(u_i, v_i)\sin\beta - f'_v(u_i, v_i)\cos\beta. \tag{20}$$

It is easily seen that $\tan\gamma_i = -\mathbf{N}_t^i/\mathbf{N}_s^i$, which gives us:

$$\tan\gamma_i = f'_u(u_i, v_i)\sin\beta + f'_v(u_i, v_i)\cos\beta, \quad i = 0, 1. \tag{21}$$

To find the top point of the curve, we solve the equation $p'(t) = \tan\alpha$, the solutions of which are:

$$t_2 = \frac{-b \pm \sqrt{b^2 - 3a(c - \tan\alpha)}}{3a}. \tag{22}$$

Under the conditions stated, there is exactly one solution t_2 to (22) satisfying $t_0 < t_2 < t_1$. After finding t_2, one could use $p(t)$ or the given 2-variable function to find the top of the ridge. The actual function would probably be more accurate.

Naturally, this method is an approximation and a more accurate method would be to use some numerical technique. For example, the method described above could be iterated as follows: After finding t_2, one could check the size of the dot product of the view vector and the normal at that point; if the size is greater than some user-defined tolerance, then the process is repeated with t_2 replacing t_0 or t_1 according the normal's direction. However, for most practical applications, a single iteration is probably sufficient.

For perspective projections, the calculations would be somewhat different. First, it would be necessary to represent the normal vectors $\mathbf{N}^i$ in the form $\mathbf{N}^i = \mathbf{N}_\Pi^i + \mathbf{N}_+^i$, where $\mathbf{N}_\Pi^i$ is in Π and $\mathbf{N}_+^i$ is orthogonal to Π. This is done by using the known vector normal to Π, which is just (A, B, C)-see (8). Only the $\mathbf{N}_\Pi^i$s are needed since the spline lies entirely in Π. The view vector should be taken from the surface point toward the viewpoint.

5.3 Correction of the "Bump" Problem

As mentioned, this problem occurs with intensity interpolation shading when a ridge or valley is viewed from above. When a ridge is directly in front of the viewpoint, as in Fig. 3, it can be detected simply by observing that two successive values of $\mathbf{L}\cdot\mathbf{N}_\Pi$ have opposite signs. In our implementation, the light source and viewpoint coincide, so $\mathbf{L} = \mathbf{V}$, where $\mathbf{V}$ denotes the vector toward the viewpoint. If $\mathbf{L} \neq \mathbf{V}$, just replace $\mathbf{V}$ by $\mathbf{L}$ in the following. As in Section 5.2, $\mathbf{N}_\Pi$ is the component of $\mathbf{N}$ lying in Π.

Fig. 5 demonstrates the bump problem. We can see that the problem cannot be detected in the same way as the ridge problem because the values of $\mathbf{V}\cdot\mathbf{N}_\Pi$ at two successive points have the same sign. From the figure, we can see that the cross (vector) products of $\mathbf{V}$ and $\mathbf{N}_\Pi$ are in opposite directions, so their dot product is negative. In other words, the test for a bump situation is $(\mathbf{V^0}\times\mathbf{N_\Pi^0})\cdot(\mathbf{V^1}\times\mathbf{N_\Pi^1}) < 0$, where $\mathbf{V^i}, \mathbf{N_\Pi^i}$ are, respectively, the view and normal vectors (in Π) at the two successive points.

The bump problem can be corrected in two ways: One of them is to fit a cubic spline as described above and to take the extreme point of the spline as an additional sample point. Another method is to sample the region between the two points at a higher frequency. Normal interpolation shading completely avoids this problem because, at each pixel, the intensity is calculated according to some value of the normal, interpolated between $\mathbf{N}^0$ and $\mathbf{N}^1$.

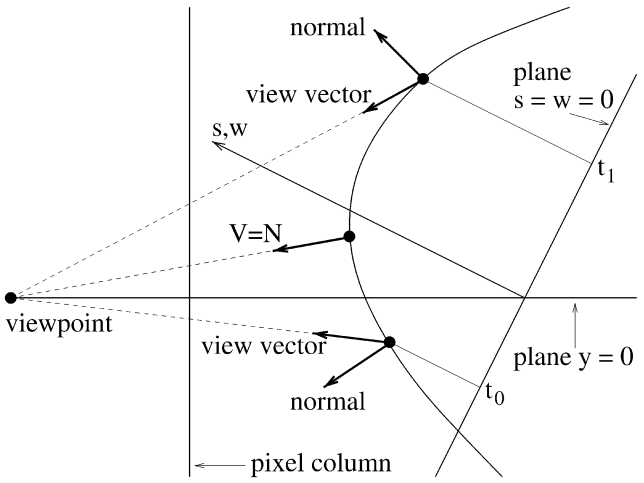

Fig. 5. A "bump" viewed from above.

6 Extensions and Applications

In this section, we describe some further extensions of the floating column algorithm. These extensions have not been implemented, but the explanations are sufficiently detailed for the graphics practitioner.

6.1 Graphics Hardware Utilization and Implementation

As mentioned, the floating column produces strips of pixels with all the information for intensity interpolation shading, so this can be done with graphics hardware. If the hardware Z-buffer is used, there is no need to maintain the minimum and maximum at every step. Normal interpolation shading can also be done with state-of-the-art graphics cards, such as the NVIDIA GeForce2 Ultra [3] with its "per-pixel" shading. As mentioned in Section 2, the algorithm can be implemented on a graphics card and several of the card's processors can execute it in parallel.

If a Z-buffer is used, other objects, besides the function surface, can be rendered. If we want to "place" an object on the surface, all we need are its u, v coordinates and the object can then be placed in position (u, w, v), where $w = f(u, v)$ and f is the displayed function. Combining this with texture mapping (see Section 6.7), we can display functionally defined surfaces with texture and various objects scattered on the surface. The function can be modified instantly and the objects will always be at the correct height. The above can be used to obtain instantly modifiable functionally defined terrains, for use in games or flight simulators. Another use for a modifiable surface is the animated display of a liquid surface (or any time-varying function)—all we need is a function $f(u, v, t)$, which provides the height as a function of position and time.

6.2 Shadows

The ability to produce shadows is important for realistic terrain visualization. We propose two approaches to this problem. One is a preprocessing approach that creates a

shadow map on the terrain and another is an approach that generates shadows on the fly. Both methods are based on our version of the classical floating horizon algorithm, as presented in Section 4.1. We assume the terrain can be displayed by the floating column, as described in Section 2.

6.2.1 Shadow Textures

A *shadow texture*, created for each light source, is a two-dimensional pixel array superimposed on the function (or terrain) domain. Every pixel contains just one bit, indicating whether it is in shadow or not. The idea originates from Willams' technique of creating shadow maps—see Williams [14] or the description in Foley et al. [4, Section 16.4.4]. Williams' method is to create a Z-buffer around the light source, with the light itself acting as the center of projection. Our method maps the shadow information on the terrain domain, as described below.

First, we determine the resolution of the shadow texture that we wish to impose on the terrain domain. Now, consider Fig. 2 with the light source in place of the viewpoint and some plane in place of the screen. We do not have actual pixel columns, but we need to ensure that the minimal distance between successive positions of the plane Π will guarantee that every shadow pixel will be cut by some plane; this is easily calculated from the resolution of the shadow texture. For every position of Π, consider the line formed by Π and the plane $w = 0$. The sampling plane $t = \text{const.}$ is advanced along this line from one shadow pixel to the next by using the standard DDA algorithm [4, Section 3.2.1]. If the height of the sampled point is above the current maximum for the column (or below the current minimum), then the pixel is lit and we update the extremum; otherwise, the pixel is in shadow.

When rendering the terrain itself, we again consider the line formed by the intersection of the plane Π and the u, v plane. This line intersects the pixels of the shadow texture (as described above). Thus, the line is partitioned into disjoint segments of alternating light and shadow. When rendering the pixel column, we now advance the sampling plane either to its next usual position, or to the next point of change between light and shadow, whichever is closer. This ensures that, between every two successive positions of the sampling plane, the surface is entirely lighted or entirely in shadow. This condition allows us to use interpolation shading for the entire span between the two positions of the sampling plane.

6.2.2 On-the-Fly Shadow Generation

The creation of the shadow texture needs to be carried out before visualization, but this may not be very practical for some applications. For example, the terrain might be very large and we need to see only a relatively small part of it or we may want to change the position of the light source(s) frequently. This calls for an alternative method of generating shadows, which can be carried out "on-the-fly," i.e., the shadows are calculated at the same time as the surface is rendered.

Our technique is demonstrated with the aid of Fig. 6, which shows, as an example, a circular region of interest in the domain u, v. Consider the current plane Π: We wish to determine, along a sequence of points $(u_1, v_1), \ldots, (u_n, v_n)$ on the intersection of Π and the u, v plane, which one is lighted and which is in shadow. For each such point (u_i, v_i), this can be done by our version of the floating horizon algorithm by advancing from the light source toward (u_i, v_i) while maintaining the minimum and maximum in the usual manner.

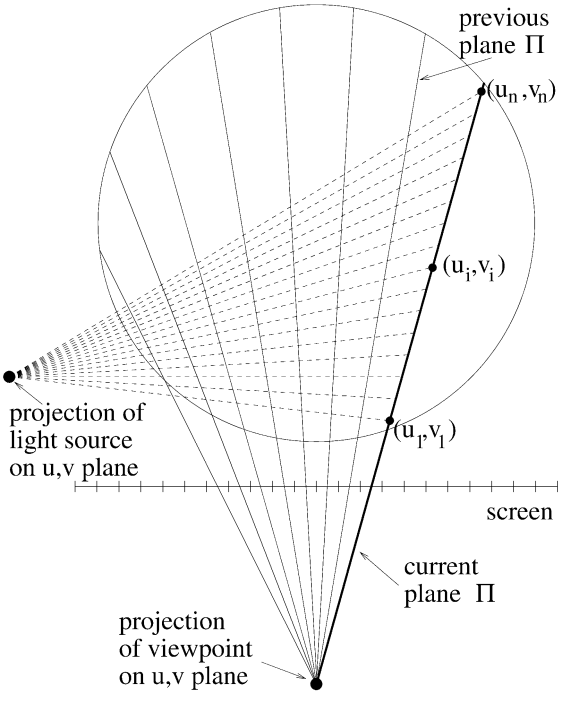

Fig. 6. Shadow generation.

It can also be observed from Fig. 6 that there is a more time-efficient method: Suppose the sequence of successive planes Π is chosen in increasing distance from the light source. Then, for each Π, the required minimum and maximum elevation values (relative to the light source) have already been computed for the previous Π. These can be stored in $O(n)$ memory and used for computing the shadows along Π. In other words, the sequence of Π-planes are used as the *sampling* planes with respect to the light source. The number of planes chosen through the light source and their precise angle can be easily calculated from the geometry of the region, the viewpoint, the light source, and the required minimal sampling distance along each Π.

6.3 Locally Adaptive Sampling

Due to the independent column mode, it is possible to modify the sampling rate in different regions of the u, v domain, according to various conditions. Supersampling has already been mentioned, but one should note that supersampling is not necessarily only for artifact correction, it can also be applied when local features require a higher sampling rate. This can occur either because a displayed function oscillates at a high frequency in a certain region or because a displayed terrain has certain ground features that require detailed sampling for proper visualization.

Locally adaptive sampling can also be applied in the opposite direction, i.e., the distance between successive sampling planes can be increased. For function visualization, this could be useful for regions where the surface normal

has relatively little variance. With terrains, this can be applied to regions that are far away, providing a simple means of level-of-detail adaptation.

6.4 Antialiasing

We consider the approach of supersampling and averaging for handling the aliasing problem. The independent column mode allows us to proceed as follows: We generate just a few (two to four) columns of the supersampled image and then perform an averaging operation to obtain one column of the final image. The first supersampled column is then abandoned and its space is used for the next supersampled column. In this manner, only $O(n)$ space is required, where n is the number of pixels in a column.

For displaying functions, we can do supersampling along the columns as described above, but, along each column, we do not really need to increase the number of pixels. Aliasing along a column occurs only at pixels which have been set by a local minimum or maximum and this information can be maintained by using floating-point numbers for the extrema. Whenever the surface reappears beyond an extremum, the pixel can be colored by a weighted average of the previous value and the new value, based on the relative pixel coverage.

6.5 Transluscency and Multiple Surfaces

A simple extension of the floating column algorithm is to incorporate transluscency. Recall that we operate on a single column at any given time, so all the data structures relate to a single column. Suppose that, for every pixel at the currently rendered column, we maintain a linked list containing the depth values (and other shading information) of all points of the surface intersected by a ray through the pixel. These depth values can be easily evaluated by interpolation in the same way as described for intensity and the front-to-back traversal will produce the linked list in increasing depth values.

After an entire column of pixels has been sampled by all the sampling planes, we can use the depth values to render the surface as if it were transluscent. At every pixel, we can blend the intensities at different distances according to any desired weighting consideration. We can even vary the color according to increasing distance. The space complexity of this process is $O(nk)$, where n is the number of pixels per column and k is the number of sampling planes.

An extension of the above procedure can be used to render two (or more) transluscent surfaces as follows: Suppose that, for a given column, we have evaluated the pixel lists for each surface as described above. For every pixel in the column, its lists are ordered by increasing distance, so these lists can be merged into one ordered list and displayed according to any desired procedure. For example, each surface can be assigned a different basic color (red, green, etc.), with its intensity depending on the number of surfaces hiding it and the resulting basic intensities can then be blended.

6.6 Back-to-Front Traversal and Alpha-Blending

One interesting difference between the monochromatic floating horizon algorithm and the shaded floating column algorithm is that the latter need not necessarily proceed in a front-to-back order. In fact, if the sampling planes are taken in *decreasing* distance from the viewpoint, we do not even need to maintain the minimum and maximum along a column; all we need is for each new span of pixels to overwrite whatever was held in those pixels. This is somewhat disadvantageous in general because many pixels might get written many times, instead of just once, as in the front-to-back order.

However, there is an advantage in this approach if we wish to render transluscent surfaces. We do not need to maintain a list of all the shading values as described above. Instead, we can render each pixel by alpha-blending its new value with the old: Suppose the current pixel value is I_0 and a new value of I_1 is evaluated, then the pixel is set by the value $\alpha I_1 + (1-\alpha)I_0$, where $0 < \alpha < 1$ is specified by the user. This new value may later be blended with a value of a closer portion of the surface. Modern graphics equipment even performs such operations in hardware.

The above method can be extended to handle multiple surfaces as follows: At each position of the sampling plane, each surface contributes one span of pixels that needs to be set. If all these spans are disjoint, then the pixels are set by blending the new values with the old, as described above. Otherwise, for each pixel belonging to two (or more) spans, we perform the above procedure for the surfaces mapped onto the pixel in a back-to-front order. This means that the distance of each surface from the pixel should be maintained, but, as noted in Section 6.5, this is easily done by interpolation.

6.7 Texture Mapping

In order to enable texture mapping, the user has to set up the texture map so that, for each u, v in the function domain, we can evaluate some texture value $T(u, v)$ with which to modify the display. This texture map may or may not depend on the displayed height field, depending on the application. Another issue related to textures is the required sampling rate; clearly, some textures will require more sampling than others. The texture can also be supplied in a similar manner to the shadow texture described in Section 6.2.

Texture mapping would probably be best handled in hardware via OpenGL: All that is required is to supply the appropriate OpenGL routines with the data at the endpoints of the displayed strip (which is an elongated patch) and the corresponding points in the u, v plane. To quote Watt and Watt [13, p. 185]: "If the object is represented parametrically, texture mapping is straightforward, since a parametric patch, by definition, already possesses u,v-values over its surface."

7 Results

The function $w = 3\sin(u\cos v)$ is used to demonstrate the new method. Fig. 7 shows the result of our version of the regular floating horizon algorithm, using perspective projection. Fig. 8 demonstrates the result of the floating column algorithm, with a sampling rate identical to that of Fig. 7. We can see two kinds of artifacts. The most noticeable problem is the shading artifact that appears as triangular patches near the top of ridges. Note that the boundaries

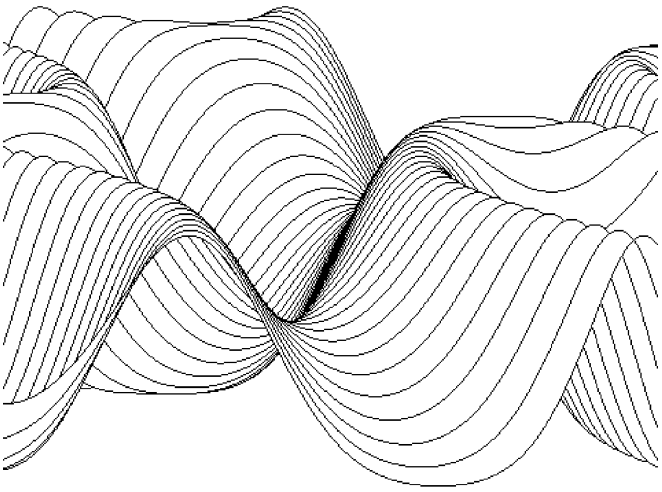

Fig. 7. The modified floating horizon algorithm.

between adjacent triangles occur at the exact positions of the lines in Fig. 7. The second problem also appears near ridges, where the sampling planes form an oblique angle with the top of the ridge. This results in an artificial corrugated appearance of the ridge—see especially the top left part of Fig. 8. The corrugated top also appears with the original algorithm, but it is acceptable for line drawings.

Fig. 9 shows the result when local supersampling is used to correct the artifacts, with intensity interpolation shading. Supersampling was used when the following problems were detected: a ridge (or valley), too great a difference in intensity, and bumps. Supersampling was done by sampling at 16 times the frequency of the regular sampling rate. Other rates are also possible, as well as an adaptive sampling rate. Supersampling for the shading artifacts was implemented whenever the difference in intensity was found to be greater than $1/16$ of the maximal intensity; this value was found to produce good results.

Figs. 10 and 11 show an orthographic projection of the function surface using normal interpolation shading,

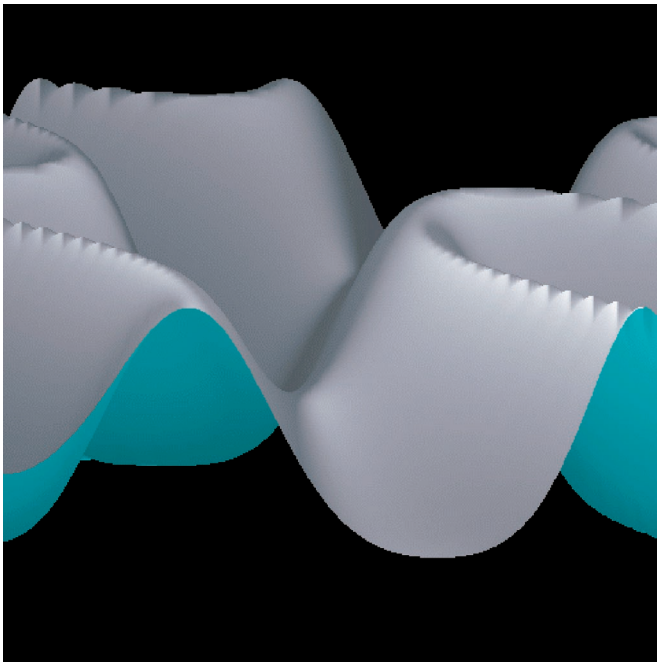

Fig. 8. Intensity interpolation shading—no artifact corrections.

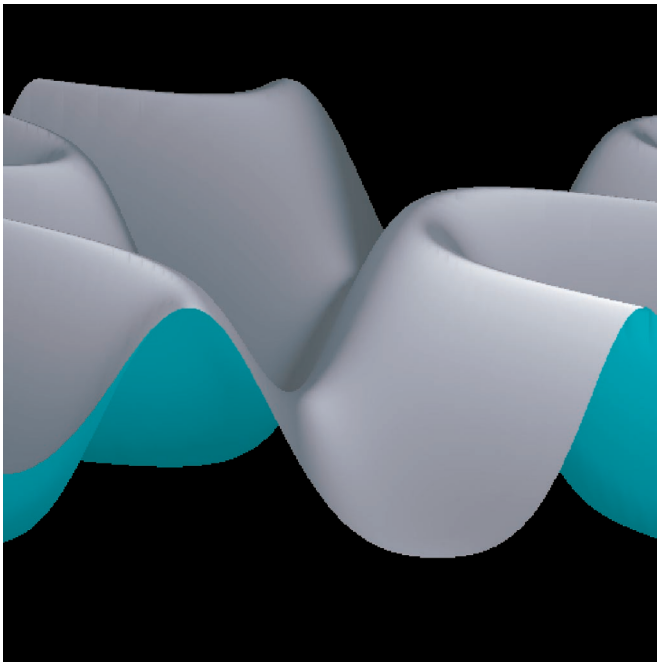

Fig. 9. Intensity interpolation shading—artifacts corrected by local supersampling.

Fig. 10. Normal interpolation shading—artifacts corrected by fitting local splines.

Fig. 11. Normal interpolation shading—artifacts corrected by supersampling.

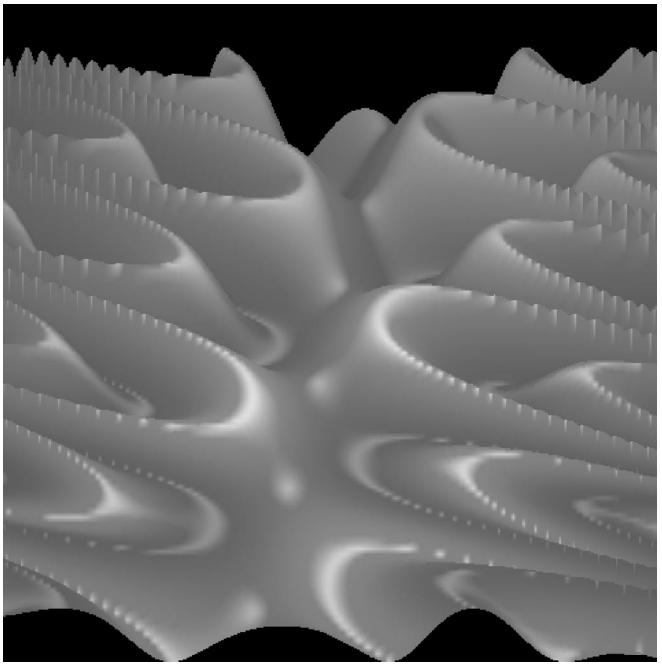

Fig. 13. Bird's eye view showing intensity discrepancies at local extrema viewed from above.

without shading corrections. In Fig. 10, the corrugated ridges were dealt with by fitting a local cubic spline and, in Fig. 11, the ridges were handled with local supersampling. Both methods are a great improvement over the original problem, but neither is quite as good as the intensity interpolation shading with shading corrections. Of course, shading corrections can also be done with normal interpolation.

Figs. 12, 13, and 14 show a perspective projection of the same function surface, viewed from the same angle as the previous figures. The surface is viewed from a greater distance and using a wider viewing angle (smaller focal distance d_2—see Fig. 2) and with a larger extent of the sampled function space. Note that the line drawing provides very few visual cues for the portion of the surface that is viewed from above, i.e., the bottom part of Fig. 12. Figs. 13 and 14 were rendered with intensity interpolation shading and the bump problem can be seen in the lower half of Fig. 13. In Fig. 14, the artifacts were corrected with local supersampling.

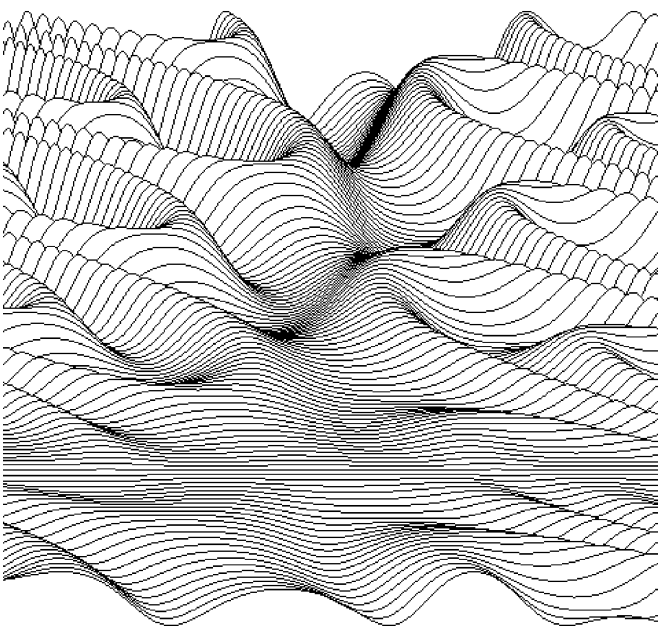

Fig. 12. Perspective, bird's eye view of the function surface.

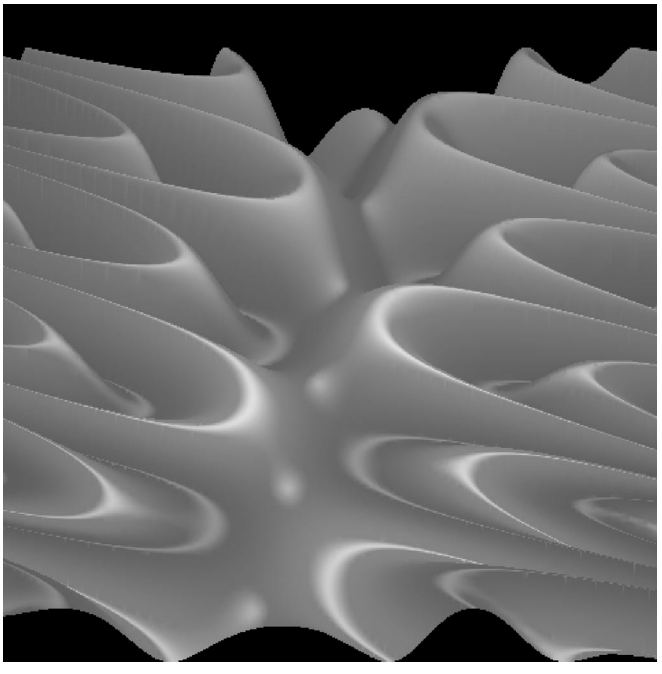

Fig. 14. Bird's eye view—artifacts corrected by local supersampling.

TABLE 1
Statistics for Images on Alphastation; Resolution 480×477

Figure no. and display method	No. of samples	Time (sec.) with interpolations	Time (sec.) w/out interpolations
7, line, perspective	25281	< 0.0167	< 0.0167
8, iis, no corrections	25281	0.0332	0.0167
9, iis, supersampling	86957	0.0833	0.0666
10, nis, spline corrections	25281	0.0500	0.0333
11, nis, supersampling	61149	0.0666	0.0500
12, line, perspective	61533	0.0167	0.0167
13, iis, no corrections	61533	0.0666	0.0500
14, iis, supersampling	289575	0.2666	0.2500
Notes: *line*: line drawing using the modified floating horizon algorithm. *iis, nis*: shaded floating column using intensity & normal interpolation shading, resp.			

Timing experiments were done to compare the runtimes of the floating column algorithm with the floating horizon algorithm, to compare the different artifact correction methods, and to examine the consequences of relegating all interpolation calculations to hardware. These times include only the computation times, since different workstations will have widely varying graphics capabilities, but the computation times can be expected to exhibit the same relative runtime ratios. Table 1 shows the total number of samples evaluated for each image and runtimes on a Digital Alpha workstation with an EV67 processor running at 667 MHz. Shown in the table are the timings for software-computed interpolations and the timings without the interpolation calculations (assuming these are relegated to the graphics hardware). All the images were done with a resolution of 480 lines by 477 pixel columns.

The floating column is, of course, slower than the floating horizon algorithm, but today's processors are many orders of magnitude faster than those originally used for the floating horizon and the superior results more than justify the use of the new method. The timings for supersampling can be further reduced (by approximatey 50 percent) simply by reducing the supersampling rate from 16 to 8 or less and setting a larger threshold value for shade differences (e.g., from the current 0.0625 to 0.125). Adaptive supersampling would undoubtedly yield better results, but it has not yet been implemented.

Regarding the artifact correction methods, normal interpolation shading with spline fitting, with all interpolations done by the hardware, is the fastest, but the differences are not great on fast state-of-the-art workstations. As mentioned in Section 6.1, normal interpolation shading can be done by graphics hardware. Even though normal interpolation with spline-fitting requires more calculations than intensity interpolation (without supersampling), it avoids the need for supersampling, hence its overall efficiency.

8 Conclusions

The floating column algorithm is a new method for the visualization of function surfaces. It is basically a simple method that operates in constant space and does not require surface patches. The independent column mode allows local operations to enhance the display and correct some artifacts. The new technique is inherently parallel and, thus, suitable for the new generation of multiprocessor workstations. It requires constant space, so it can be implemented in a graphics card, where it can run in parallel on several of the card's processors. It is also useful when memory is limited, e.g., in hand-held devices.

The new algorithm can be applied to functions of two variables, replacing the old floating horizon algorithm with a shaded version which is more appropriate for today's high quality graphics displays. Both intensity interpolation and normal interpolation shading are supported and both methods produce pixel-wide strips which can be displayed by graphics hardware with texture mapping. The floating column can sample the function at different frequencies in different regions, according to the local features or local oscillation rate. Furthermore, compared to patching techniques, interactive modification of the function is immediate since only a redefinition of the function is required.

The new technique is applicable to terrains without the need for flat polygon patching, provided there is some method of associating a height with every domain point. Shadows are possible, as well as local changes in the sampling rate according to local features or distance, thus providing a simple level-of-detail adaptation. With a hardware Z-buffer, various objects can be placed automatically on the surface. These features, together with the inherent parallelism, provide a potential for high-quality real-time rendering of terrains on multiprocessor workstations. This could be useful for virtual reality applications in games or flight simulators.

The algorithm can be used for the efficient animated display of time-varying surfaces, provided a function describing the surface as a function of time is supplied. The independent column mode also enables efficient antialiasing by supersampling and averaging: Only two to three supersampled columns need to be stored at any given time to obtain one column of the final image.

Appendix A
Detailed Algorithms

We present the pseudocode for our modified floating horizon algorithm and the floating column algorithm. Text following the symbol # is a comment.

A.1 Modified Floating Horizon (Monochromatic)

```
fhr[Max_Columns] # array of structures, each
# containing integers ymin, ymax, yprev for
# minimum, maximum and previous values of y.
# Determine values of necessary constants
# (implementation dependent):
int bot_row, top_row # bottom and top scanlines
int Midx, Midy # coordinates of central pixel
real Radius  # radius in u,v plane limiting
             # the displayed region
real Ratio # ratio of units in function
           # domain and screen coordinates
real Sina, Sinb, Cosa, Cosb
           # sin and cos of alpha and beta
# Geometric variables - see text:
real x0, y0, y1, z1, t1, r1, u1, v1, w1,
d1, d2, d3 = d1 + d2
# determine first and last pixel column:
first_col = Midx + (int)(-RADIUS*Ratio)
last_col = Midx + (int)(RADIUS*Ratio)
for (col = first_col, col <= last_col,
   col++) # initialize min and max in column
  {fhr[col].ymin = MAXINT
   fhr[col].ymax = -MAXINT}
for (t1 = first_t, t1 <= last_t, t1 += t_step)
  {# loop on horizons. precompute constants:
   real t1sb = t1*Sinb t1cb = t1*Cosb
   real t1sa = t1*Sina t1ca = t1*Cosa
   for (col = first_col, col <= last_col,
        col++) # loop on columns
     {x0 = (col - Midx)/Ratio
      r1 = x0*(t1ca + d3)/d1
      u1 = r1*Cosb + t1sb
      v1 = -r1*Sinb + t1cb
      w1 = Func(u1,v1) # function value
      y1 = w1*Cosa - t1sa
      z1 = w1*Sina + t1ca
      if (z1 <= -d2) continue
       # cannot display this point
      y0 = y1*d1/(z1+d3)
      row = (int)ROUND(y0*Ratio + Midy)
       # scanline of pixel
      if (row < bot_row) row = bot_row-1
      if (row > top_row) row = top_row+1
       # don't waste time on invisible pixels
      if (fhr[col].ymin==MAXINT &
          fhr[col].ymax==-MAXINT)
        {# First pixel in column:
         fhr[col].ymin = fhr[col].ymax
           = fhr[col].yprev = row
         if (col == first_col)
           # check if first column
           Putpixel(col, row)
         else # draw segment for continuity
           Line (col-1, fhr[col-1].yprev,
                 col, row)
        }
      else  # not first pixel in column,
            # check if visible:
        {if (row < fhr[col].ymin)
           # below min in column
         {fhr[col].ymin =
           fhr[col].yprev = row
          if (col == first_col)
           # check if first column
           Putpixel(col, row)
          else # draw segment for continuity
           Line (col-1, fhr[col-1].ymin,
                 col, row)
         }
        elseif (row > fhr[col].ymax)
         # above max in column
         {fhr[col].ymax =
           fhr[col].yprev = row
          if (col == first_col)
           # check if first column
           Putpixel(col, row)
          else # draw segment for continuity
           Line (col-1, fhr[col-1].ymax,
                 col, row)
         }
        if (col > first_col)
         # check if partial line needed:
         if (fhr[col-1].yprev > fhr[col].ymax
           & row <= fhr[col].ymax)
          Line (col-1, fhr[col-1].yprev,
                 col, fhr[col].ymax)
         elseif (fhr[col-1].yprev <
                 fhr[col].ymin &
                 row >= fhr[col].ymin)
           Line (col-1, fhr[col-1].yprev,
                 col, fhr[col].ymin)
       } # end else not 1st pixel in column
     } # end of loop on columns
  } # end of loop on horizons
} # end monochromatic floating horizon algorithm
```

A.2 Floating Column Algorithm (Shaded)

For the sake of simplicity, we present the floating column algorithm with intensity interpolation shading. The changes for normal interpolation are obvious.

```
# Determine necessary constants
# (implementation dependent):
int bot_row, top_row # bottom and top scanlines
int Midx, Midy # coord. of central pixel
real Radius   # radius in u,v plane limiting
              # the displayed region
real Ratio # ratio of units in function
           # domain and screen coordinates
real Sina, Sinb, Cosa, Cosb
           # sin and cos of alpha and beta
real basic_step # basic step size
int top_color, bot_color
# colors for top and bottom of surface
```

```
# Geometric variables - see text:
real x0, y0, y1, z1, t1, r1,
u1, v1, w1, d1, d2, d3 = d1 + d2
struct Vector{real x, y, z}
Vector L, N # structures for vectors L and N
Vector LxN # cross product of L and N Vector
LxN_old # old value of LxN
real LdotN, LdotN_old
  # current and previous values of L.N
real dist
  # distance of displayed point to screen
int ymin, ymax, yprev
  # minimum, maximum and previous values of y
# determine first and last pixel column:
first_col = Midx + (int)(-RADIUS*Ratio)
last_col = Midx + (int)(RADIUS*Ratio)
for (col = first_col, col <= last_col, col++)
  {# loop on columns:
   ymax = yprev = MAXINT
   ymin = -MAXINT
   LdotN_old = 0.
    # to avoid false ridge at start
   x0 = (col - Midx)/Ratio
   t1 = first_t t_step = basic_step
   while (t1 <= last_t) # loop on horizons:
    # step size may vary due to supersampling
   {r1 = x0*(t1*Cosa + d3)/d1
    u1 = r1*Cosb + t1*Sinb
    v1 = -r1*Sinb + t1*Cosb
    w1 = Func(u1, v1) # function value
    y1 = w1*Cosa - t1*Sina
    z1 = w1*Sina + t1*Cosa
    if (z1 <= -d2)
      # cannot display point - go to next t1
      {t1 += t_step, continue}
    y0 = y1*d1/(z1+d3)
    row = (int)ROUND(y0*Ratio + Midy)
      # scanline of pixel
    dfdu = Func_u(u1, v1)
    dfdv = Func_v(u1, v1)
      partial u- and v-derivatives
    N.x = -dfdu*Cosb + dfdv*Sinb
    N.y = dfdu*Sasb + Cosa + dfdv*Sacb
    N.z = -dfdu*Casb + Sina - dfdv*Cacb
    Normalize(N)
    L.x = -x1, L.y = -y1, L.z = -(D3 + z1)
    Normalize(L)
    LdotN = Dot(L, N) # dot product
    LxN = Cross(L, N) # cross product
    dist=sqrt(SQ(x0-x1)+SQ(y0-y1)+
                SQ(z0-z1))
    if (Intensity_Interpolation)
      intens = Intensity(fabs(LdotN), dist)
    if ( Problem() )
    # check if problem correction required
      {Problem_Correction()
       t1 += t_step
       continue # to next horizon
      }
    if (yprev = MAXINT)
      {# First pixel in column:
       if ((LdotN >= 0.) color = top_color
       else color = bot_color
       Putpixel(col, row, intens, color)
       ymin = ymax = row
      }
    else # not first pixel in column
      {# check if visible:
       if (row < ymin) # below min for column
        {# fill pixels by interpolation:
         Interpolate(col, ymin-1, row, yprev,
          intens_old, intens, bot_color)
         ymin = row
        }
       elseif (row > ymax) # above max for column
        {# fill pixels by interpolation:
         Interpolate(col, ymax+1, row, yprev,
          intens_old, intens, top_color)
         ymax = row
        }
      }
    # save current values for next horizons:
    intens_old = intens
    yprev = row  # saved even if surface
                 # point was not visible
    LdotN_old = LdotN
    LxN_old = LxN
    # required only for certain "problems"
   } # end of loop on horizons
  } # end of loop on columns
} # end of shaded floating column algorithm

function Problem()
# Function to determine if there is a special
# problem that needs handling. Any combination
# of the following tests can be done:
# 1. Current and previous surface normals in
    # opposite direction relative to viewer:
    # if ( LdotN * LdotN_old < 0. )
# 2. Difference in intensity greater than
    # user-defined threshold:
    if (|intens - intens_old| >
        Intens_Threshold)
# 3. Two successive horizons straddle a ridge or
    # valley, but both points are visible
        (relevant
    # only for intensity interpolation shading,
    # which can miss the highlight at the
         extremum):
    if ( Dot(LxN, LxN_old) < 0. )

procedure Problem_Correction()
# This procedure depends on the type of problem
# correction that the user wishes to implement.
# There are two main correction procedures:
# Supersampling: execute the main loop from
```

```
# previous value of t1 to current t1, but with
# a smaller step size (user defined). Spline
# fitting: based on old and new values, compute
# a cubic spline between current point and
# previous one. Find extremum of spline and
# execute main horizon loop from previous point
# to extremum. Continue main loop with extremum
# serving as previous point.

procedure Interpolate(column, start_row,
   end_row, yprev, intens_old, intens, color)
# This procedure fills all the pixels in the
# given column, from start_row to end_row.
# The pixels are filled with the given color,
# with their intensity interpolated linearly
   from
# intens_old at row yprev to intens at row
   end_row.
```

ACKNOWLEDGMENTS

The author is indebted to the anonymous reviewers whose detailed comments have led to an improved presentation of the paper. The runtime experiments were performed on equipment funded by grant no. 19050101311 of the Israel Ministry of Science.

REFERENCES

[1] F. Albersmann, H. Müller, F. Weller, and A. Zabel, "Efficient Direct Rendering of Digital Height Fields," *Proc. IFI TC5/WG5.10 and CSI Int'l Conf. Visual Computing (ICVC '99),* S.P. Mudur, D. Shikhare, J.L. Encarnacao, and J. Rossignac, eds., pp. 44-52, Feb. 1999.

[2] D. Cohen-Or, E. Rich, U. Lerner, and V. Shenkar, "A Real-Time Photo-Realistic Visual Flythrough," *IEEE Trans. Visualization and Computer Graphics,* vol. 2, no. 3, pp. 255-265, Sept. 1996.

[3] NVIDIA Corp., "Technology Brief," technical report, http://www.nvidia.com, 2000.

[4] J.D. Foley, A. van Dam, S.K. Feiner, and J. Hughes, *Computer Graphics: Principles and Practice,* second ed. Reading, Mass.: Addison-Wesley, 1990.

[5] G. Glaeser, *Fast Algorithms for 3D-Graphics.* New York: Springer-Verlag, 1994.

[6] H. Hoppe, "Progressive Meshes," *Computer Graphics Proc.,* pp. 99-108, Aug. 1996.

[7] H.W. Kohl, "Hidden-Curve Algorithm for Correct Grid Surface Representation of Functions of Two Variables," *Computers & Graphics,* vol. 20, no. 3, pp. 243-261, 1996.

[8] A. Lamothe, "Real-Time Voxel Terrain Generation," *Game Developer,* vol. 4, no. 8, pp. 34-44, Nov. 1997.

[9] C.K. Pokorny and C.F. Gerald, *Computer Graphics: The Principles Behind the Art and Science.* Irvine, Calif.: Franklin, Beedle & Assoc., 1989.

[10] A. Ralston and P. Rabinowitz, *A First Course in Numerical Analysis,* second ed. New York: McGraw-Hill, 1978.

[11] D.F. Rogers, *Procedural Elements for Computer Graphics.* New York: McGraw-Hill, 1985.

[12] S.L. Watkins, "Algorithm 483: Masked Three-Dimensional Plot Program with Rotations," *Comm. ACM,* vol. 17, no. 9, pp. 520-523, Sept. 1974.

[13] A. Watt and M. Watt, *Advanced Animation and Rendering Techniques.* Reading, Mass.: Addison-Wesley, 1994.

[14] L. Williams, "Casting Curved Shadows on Curved Surfaces," *Computer Graphics Proc.,* pp. 1-11, Aug. 1978.

[15] H. Williamson, "Algorithm 420: Hidden-Line Plotting Program," *Comm. ACM,* vol. 15, no. 2, pp. 100-103, Feb. 1972.

[16] T.J. Wright, "A Two-Space Solution to the Hidden Line Problem for Plotting Functions of Two Variables," *IEEE Trans. Computers,* vol. 22, no. 1, pp. 28-33, Jan. 1973.

Dan Gordon received the BSc and MSc degrees in mathematics from the Hebrew University of Jerusalem and the DSc degree in mathematics from the Technion-Israel Institute of Technology. He is an associate professor in computer science at the University of Haifa and has held visiting positions in Israel and the USA. His research interests include computer graphics, visualization, medical image reconstruction, and processor arrays. He is a member of the ACM and ACM-SIGGRAPH.