Efficient Algorithms for 3D Scan-Conversion of Parametric Curves, Surfaces, and Volumes

Arie Kaufman

Department of Computer Science
State University of New York at Stony Brook
Stony Brook, NY 11794-4400

Abstract

Three-dimensional (3D) scan-conversion algorithms, that scan-convert 3D parametric objects into their discrete voxel-map representation within a Cubic Frame Buffer (CFB), are presented. The parametric objects that are studied include Bezier form of cubic parametric curves, bicubic parametric surface patches, and tricubic parametric volumes. The converted objects in discrete 3D space maintain pre-defined application-dependent connectivity and fidelity requirements.

The algorithms introduced here employ third-order forward difference techniques. Efficient versions of the algorithms based on first-order decision mechanisms, which employ only integer arithmetic, are also discussed. All algorithms are incremental and use only simple operations inside the inner algorithm loops. They perform scan-conversion with computational complexity which is linear in the number of voxels written to the CFB. All the algorithms have been implemented as part of the *CUBE Architecture,* which is a voxel-based system for 3D graphics.

CR Categories and Subject Descriptors: I.3.3 [**Computer Graphics**]: Picture/Image Generation; I.3.5 [**Computer Graphics**]: Computational Geometry and Object Modeling; I.3.7 [**Computer Graphics**]: Three-Dimensional Graphics and Realism.

General Terms: Algorithms, Computer Graphics.

Additional Key Words and Phrases: Bezier curves, Bezier surfaces, Bezier volumes, cubic frame buffer, three-dimensional scan conversion, voxel.

This work was supported by the National Science Foundation under grant DCR-86-03603.

 ACM-0-89791-227-6/87/007/0171 $00.75

1. Introduction

Three primary representation domains are common for graphical data: the *continuous geometric model,* the *discrete pixel image,* and the *discrete voxel image.* The former two representations have received considerable attention in the literature, and there is an abundance of 2D scan-conversion algorithms which are fundamental to raster systems. Such an algorithm converts a 2D continuous geometric representation into a set of pixels in the pixel-image plane. Recent papers [16, 18] have introduced and developed the notion of a *three-dimensional (3D) scan-conversion algorithm,* which converts a 3D analytic representation into a set of voxels in the discrete voxel domain. We focus here on 3D scan-conversion algorithms for the following 3D parametric objects:

- Cubic parametric curves
- Bicubic parametric surface patches
- Tricubic parametric volumes

There are four voxel-image space architectures, CUBE [13-15], GODPA [11], PARCUM [12], and $3DP^4$ [21], and many other voxel-based software systems that have been reported in the literature. The theme of these systems is that the 3D inherently continuous scene is discretized, sampled, and stored in a large 3D cubic frame buffer of unit cells called volume elements or *voxels.* A CFB with 512^3 resolution and 8-bit-deep voxels, for example, is a large memory buffer of size 128M bytes. As computer memories are getting significantly cheaper and more compact, such huge memories and consequently voxel-based hardware systems are becoming more and more practical.

All the four voxel-based architectures utilize high performance multi-processing architecture to generate shaded projections of the 3D scene within the cubic buffer. They all assume that a database of voxel-based objects exists for loading into the CFB. Unlike the other systems that operate solely in the discrete voxel-image space, CUBE further provides object space capabilities and ways to intermix the two ([17]). The object-space capabilities are enabled using a spectrum of 3D scan-conversion algorithms [16, 18], those that pertain to parametric curves, surfaces and volumes are discussed in this paper. All these algorithms have been implemented as part of the 3D geometry processor of the CUBE architecture. They enable CUBE to cater to a wide variety of applications which accept a geometric representation of the 3D scene as well as sampled experimental data.

Although the voxel representation is more effective for empirical imagery, it also has a significant utility in synthetic 3D graphics or in applications merging empirical and synthetic images. For example, a synthetic injection needle or a scalpel can be superimposed on an ultrasound image, or a model of the skull can be overlaid, possibly with semi-transparent colors, upon a computed tomography scan of the skull. As a consequence of the proliferation of voxel-based systems, there is a growing need for mechanisms to input geometric models into such systems. The uniqueness of the CUBE architecture is in providing such a mechanism for a geometric model to be discretized and optionally overlaid upon and/or compared with the experimental data.

Terms used in the rest of the paper and the requirements from a 3D scan-converter are outlined in Section 2. The basic avenues to access the CFB are sketched in Section 3, followed by the 3D scan-conversion algorithms for curves (in Sections 4, 5 and 6), for surfaces (in Sections 7, 8 and 9), and for volumes (in Sections 10 and 11). The algorithms are described in pseudo-C where the C keywords are in **bold** face. Variables are shown in *italics* font. Comments are enclosed between a pair of double quotes and printed in "*italics*". Definitions are C language type definitions. Subscripts, superscripts and array notations are used interchangeably.

2. Terminology and Requirements

Most of the terms used in this paper for 3D discrete topology are generalizations of those used in 2D discrete topology (see e.g., [19, 22, 24, 25]). Let us mark the continuous 3D space by R^3, and the discrete 3D voxel-image space, which is a 3D array of grid points, by Z^3. We shall term a *voxel* or the *region contained by a 3D discrete point* (x, y, z), as the continuous region (u, v, w) such that $x-0.5<u \leq x+0.5$, $y-0.5<v \leq y+0.5$ and $z-0.5<w \leq z+0.5$. This assumes that the voxel "occupies" a unit cube centered at the grid point (x,y,z), and the array of voxels tessellates Z^3. Although there is a slight difference between a grid point and a voxel, they will be used interchangeably.

Each voxel $(x,y,z) \in Z^3$ has three kinds of *neighbors*:

(1) It has six *direct neighbors (face neighbors)*: $(x+1,y,z)$, $(x-1,y,z)$, $(x,y+1,z)$, $(x,y-1,z)$, $(x,y,z+1)$, and $(x,y,z-1)$.

(2) It has twelve *indirect neighbors (edge neighbors)*: $(x+1,y+1,z)$, $(x-1,y+1,z)$, $(x+1,y-1,z)$, $(x-1,y-1,z)$, $(x+1,y,z+1)$, $(x-1,y,z+1)$, $(x+1,y,z-1)$, $(x-1,y,z-1)$, $(x,y+1,z+1)$, $(x,y-1,z+1)$, $(x,y+1,z-1)$, and $(x,y-1,z-1)$.

(3) It has eight *remote neighbors (corner neighbors)*: $(x+1,y+1,z+1)$, $(x+1,y+1,z-1)$, $(x+1,y-1,z+1)$, $(x+1,y-1,z-1)$, $(x-1,y+1,z+1)$, $(x-1,y+1,z-1)$, $(x-1,y-1,z+1)$, and $(x-1,y-1,z-1)$.

We further define in Z^3 the six direct neighbors as *6-neighbors*. Both the six direct and twelve indirect neighbors are defined as *18-neighbors*. All three kinds of neighbors are defined as *26-neighbors*. A *6-connected path* is a sequence of voxels such that consecutive pairs are 6-neighbors. An *18-connected path* is a sequence of 18-neighbor voxels, while a *26-connected path* is a sequence of 26-neighbor voxels.

Three metrics on Z^3, which correspond to the three connectivity kinds, are defined. The *6-distance* between two voxels (x_1,y_1,z_1) and (x_2,y_2,z_2) is:

$$d_6 = |x_2-x_1| + |y_2-y_1| + |z_2-z_1| \quad (1)$$

which is the length of the shortest 6-connected path between them. The *26-distance* between two voxels is:

$$d_{26} = \max(|x_2-x_1|, |y_2-y_1|, |z_2-z_1|) \quad (2)$$

which is the length of the shortest 26-connected path between them. The *18-distance* between two voxels is:

$$d_{18} = \max\left(d_{26}, \left\lceil \frac{d_6}{2} \right\rceil\right) \quad (3)$$

which is the length of the shortest 18-connected path between them.

A 3D scan-converter is required to obey some fidelity, connectivity, and efficiency requirements. These requirements are met by the algorithms presented in this paper. The basic fidelity requirements in scan-converting an object from R^3 to Z^3 are:

1) The discrete points, for which the region contained by them is entirely inside the continuous object, are in the converted discrete object.

2) The discrete points, for which the region contained by them is entirely outside the continuous object, are not in the converted discrete object.

Obviously, some discrete points will not belong to either of the above cases, and more guidelines are necessary. Those are:

3) If the object is a curve (1D object), the converted object will meet certain connectivity requirements. The converted endpoints will be in the converted object.

4) If the object is a surface (2D object), it will meet certain "lack of tunnels" connectivity requirements. The converted curved "edges" will be in the converted object.

5) If the object is a volume (3D object), its "inside" will be converted according to requirement 1. Other points will be treated by majority decision - the discrete point is in the object if more than half its region is in the continuous object.

For curves we shall require 6-connectivity, 18-connectivity or 26-connectivity, depending on implementation needs. For surfaces we shall require lack of 6-connected tunnels. 18-connectivity or 26-connected tunnels can also be disallowed depending on implementation requirements. For solid volumes 6-connectivity will be required in order to avoid any internal cavities.

As we show later the computational complexity of the algorithms is equal to the number of discrete points converted multiplied by a constant. The goals are, thus, to attempt to decrease the constant, and to use simple operations within inner algorithm loops. Furthermore, we will strive to use integer arithmetic whenever possible, and fixed-point arithmetic only when impossible otherwise.

3. Cubic Frame Buffer Access

We assume here that the CFB access system of the CUBE architecture accepts the point/voxel location in three (X, Y and Z) registers, and its color in a fourth register, *COLOR*. The algorithms access the CFB using the following auxiliary functions. Each of the first 3 functions executes in one machine instruction and should therefore be fast:

1) *NEW_POS (axis, value);*
This function assigns *value* to the *axis* register (*X*, *Y*, or *Z*).

2) *UPDATE_POS (axis, increment);*
Increments (decrements) the *axis* register in one machine instruction, where *increment* is the coordinate step.

3) *PUT_VOXEL ();*
Writes the voxel, whose address is in the X, Y, and Z registers with color as in the *COLOR* register, into the CFB.

4) *WRITE_VOXEL (x, y, z, c);*
Updates the *X*, *Y*, *Z* and *COLOR* registers with the specified parameters, and then calls *PUT_VOXEL* to "put" the voxel into the CFB.

4. Parametric Polynomial Curves

A parametric polynomial 3D curve is defined as polynomials of some parameter t, one for each x, y, and z. Varying the parameter, e.g., from 0 to 1, defines all the points along the curve segment. Many systems, including ours, use polynomials of the third degree, since this is the lowest degree parametric polynomial that can describe a well-formed 3D curve with four boundary constraints. Matrix notation of the cubic polynomial of the curve f is:

$$f(t) = T\ M\ G \qquad 0 \le t \le 1 \tag{4}$$

$$T = [t^3\ \ t^2\ \ t\ \ 1] \qquad G = \begin{bmatrix} G_1 \\ G_2 \\ G_3 \\ G_4 \end{bmatrix} \tag{5}$$

the 4×4 matrix M is the geometric basis matrix, G is the geometric vector (G_i are the geometric *control points*), and $M\ G$ is the coefficients of the cubic polynomial.

Common cubic representations are *Hermite* form [7, 10], *B-Spline* form [8, 23], and *Bezier* form [1-3]. In the latter, G_1 and G_4 are the curve endpoints, while $3(G_2-G_1)$ and $3(G_4-G_3)$ are the end tangents. The matrix M for the Bezier representation is:

$$M_b = \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \tag{6}$$

In Bezier form the curve resides wholly within the convex hull of its control points G_i. In this paper, curves are assumed to be in Bezier form which is widely used in graphics and geometric design. There is no loss of generality in this assumption, since all cubic curves have their Bezier representation.

Direct evaluation of the 3D curve points from the polynomial of Equation 4 is computationally intensive, even when Horner's rule is employed. An alternative method for scan-conversion is repeated bisection of the curve until the segments reside within a single discrete point (e.g., [6, 20]). Another mechanism, which is the one explored in this paper, is using the curve's third order finite forward difference matrix for a step size ϵ along the parameter t [6, 9]. The initial difference vector Δf_0 is obtained by:

$$\Delta f_0 = \begin{bmatrix} \Delta^0 f_0 \\ \Delta^1 f_0 \\ \Delta^2 f_0 \\ \Delta^3 f_0 \end{bmatrix} = E_\epsilon\ M\ G = \begin{bmatrix} 0 & 0 & 0 & 1 \\ \epsilon^3 & \epsilon^2 & \epsilon & 0 \\ 6\epsilon^3 & 2\epsilon^2 & 0 & 0 \\ 6\epsilon^3 & 0 & 0 & 0 \end{bmatrix} M\ G \tag{7}$$

Now, it is used to repeatedly obtain the differences of the next step using the following third order DDA (Digital Differential Analyzer) calculations:

$$\begin{aligned} \Delta^0 f_{i+1} &= \Delta^0 f_i + \Delta^1 f_i \\ \Delta^1 f_{i+1} &= \Delta^1 f_i + \Delta^2 f_i \qquad 0 \le i < \frac{1}{\epsilon} \\ \Delta^2 f_{i+1} &= \Delta^2 f_i + \Delta^3 f_i \\ \Delta^3 f_{i+1} &= \Delta^3 f_i \end{aligned} \tag{8}$$

and $\Delta^0 f_{i+1}$ is the next step polynomial value. These calculations require only three additions per step per ordinate, while the alternative method of utilizing the Horner's rule requires three additions and three multiplications per step per ordinate.

5. 3D Scan-Conversion of Curves

The formal data type **curve** is:

```
typedef struct curve {
        int g [3][4];          "x, y, z for 4 control points"
        color c;               "color of curve"
} curve;
```

where the 3×4 control matrix g is the four Bezier geometric control points. The coordinates are assumed to be integers or fixed-point which can be scaled up to some finite resolution integer domain. The algorithm *CURVE*, presented in Figure 1, scan-converts a 3D third degree parametric curve in the parameter t according to the forward difference equations (Eq. 8). The only problem remaining is to calculate a step size ϵ along the parameter t so as to guarantee at least 26-connectivity, that is, guaranteeing that the first difference along any dimension is not greater than 1 in magnitude. In first approximation, this is the same as:

$$\max_{0\le t\le 1} \left(\left|\frac{dx}{dt}\right|, \left|\frac{dy}{dt}\right|, \left|\frac{dz}{dt}\right| \right) \le 1 \tag{9}$$

To find the maximum, the extrema of the second degree derivatives of the parametric equations in the range $0\le t\le1$ are found by differentiation. The maximum for each component is thus:

$$n_x = \max\left(\left|\frac{dx(0)}{dt}\right|, \left|\frac{dx(1)}{dt}\right|, \left|\frac{dx(\alpha)}{dt}\right| \right) \tag{10}$$

$$n_y = \max\left(\left|\frac{dy(0)}{dt}\right|, \left|\frac{dy(1)}{dt}\right|, \left|\frac{dy(\beta)}{dt}\right| \right) \tag{11}$$

$$n_z = \max\left(\left|\frac{dz(0)}{dt}\right|, \left|\frac{dz(1)}{dt}\right|, \left|\frac{dz(\gamma)}{dt}\right| \right) \tag{12}$$

where α, β, and γ are the t values for which the linear second derivatives of the curve equation are respectively 0, provided that $0\le\alpha,\beta,\gamma\le1$. The maximum magnitude of these extrema, $n = \max(n_x, n_y, n_z)$, is the inverse of the required algorithm step ϵ. A small term is first added to n, e.g., rounding up, to compensate for the approximation. The one time

Find maximum absolute value of dx/dt, dy/dt, dz/dt for $0 \leq t \leq 1$ (Eqs. 10-12), set n to maximum of these maxima;

$\epsilon = 1/$*CEILING* (n);

Calculate initial difference vectors $\Delta x, \Delta y, \Delta z$ by multiplying matrix E_ϵ by matrix M_b and control points g (Eq. 7);

WRITE_VOXEL (*ROUND*($\Delta^0 x$), *ROUND*($\Delta^0 y$), *ROUND*($\Delta^0 z$), *c*);

for ($t = 0$; $t \leq 1$; $t \mathrel{+}= \epsilon$) { "*follow the curve*"

$\Delta^0 x \mathrel{+}= \Delta^1 x$; "*apply Eq. 8 for x*"
$\Delta^1 x \mathrel{+}= \Delta^2 x$;
$\Delta^2 x \mathrel{+}= \Delta^3 x$;

$\Delta^0 y \mathrel{+}= \Delta^1 y$; "*apply Eq. 8 for y*"
$\Delta^1 y \mathrel{+}= \Delta^2 y$;
$\Delta^2 y \mathrel{+}= \Delta^3 y$;

$\Delta^0 z \mathrel{+}= \Delta^1 z$; "*apply Eq. 8 for z*"
$\Delta^1 z \mathrel{+}= \Delta^2 z$;
$\Delta^2 z \mathrel{+}= \Delta^3 z$;

WRITE_VOXEL(*ROUND*($\Delta^0 x$), *ROUND*($\Delta^0 y$), *ROUND*($\Delta^0 z$), *c*);

}

Figure 1: *CURVE* - Algorithm for 3D Scan-Converting Curves

floating-point computation of the step ϵ requires 9 multiplications, 9 additions, 4 divisions, 8 comparisons, and one rounding.

Algorithm time complexity is linear in the maximum magnitude n, which should come near the number of painted voxels. This is so for all curves which are not too oscillative, wavy or self-intersecting. It is reasonable to assume that the curves, or series of sub-curves, dealt with here, which are third degree Bezier polynomials, belong to this class.

Each step of the algorithm loop, i.e., each voxel written into the CFB, takes 10 floating-point additions, one floating-point test, 3 roundings of floating-point numbers to integer coordinates, and one *WRITE_VOXEL*. These heavy floating-point computations are a result of the fact that the forward differences, being the first, second, and third order slopes, are floating-point numbers or fractional binary numbers (powers of the step ϵ). These difficulties with *CURVE* computations can be overcome by converting the variables to integers. A more efficient curve algorithm, *FAST_CURVE*, which uses only integer arithmetic, is discussed in the next section.

6. Efficient 3D Scan-Conversion of Curves

The forward differences are acting as first, second, and third order DDA's, where the vector $\Delta^1 f_i$ is the first order x, y, z slopes of the curve, which is used incrementally to update the voxel position $\Delta^0 f_{i+1}$ at step $i+1$ (Equation 8). The rounded coordinates of $\Delta^0 f_{i+1}$ are then used as the three integer coordinates of the $(i+1)$-st voxel written into the CFB.

Instead, a decision process can determine at each step of the algorithm which of the 26 neighbors of the current voxel lies closer to the curve being approximated. The process selects the next voxel by considering one dimension at a time, and the decision on any dimension, i.e., no change, increment, or decrement the coordinate, is independent of the other two dimensions. The variables $\Delta^0 x, \Delta^0 y, \Delta^0 z$ are used now as the *decision variables* of the algorithm, representing the residual change in the respective coordinate.

The decision process for x, for example, is as follows. Assuming that x is the current coordinate, then if $\Delta^0 x \geq 0.5$, i.e., the curve is passing closer to $x+1$ than to x, the register X is incremented (using *UPDATE_POS* function) and $\Delta^0 x$ is decremented by 1. Similarly if $\Delta^0 x \leq -0.5$, then X is decremented and $\Delta^0 x$ is incremented. Otherwise, X and $\Delta^0 x$ are unchanged. The residual value of $\Delta^0 x$ is passed to the next step, where it is first being updated by the first order slopes. Similar decision processes hold for y and z. These decision processes enable the algorithm to avoid the rounding of the three coordinates.

Furthermore, since the same decision processes hold also after scaling up the algorithm variables by a positive value, they can be transformed to all-integer processes (cf. [5]). In order to convert the forward differences to integers and to avoid the use of the fractional step ϵ at all, the E_ϵ matrix of Equation 7 is redefined by scaling it up by the positive integer scalar $2n^3$:

$$E_n = 2n^3 E_\epsilon = \begin{bmatrix} 0 & 0 & 0 & 2n^3 \\ 2 & 2n & 2n^2 & 0 \\ 12 & 4n & 0 & 0 \\ 12 & 0 & 0 & 0 \end{bmatrix} \tag{13}$$

As a result the initial difference vector is redefined as:

$$\Delta f_0 = \begin{bmatrix} \Delta^0 f_0 \\ \Delta^1 f_0 \\ \Delta^2 f_0 \\ \Delta^3 f_0 \end{bmatrix} = E_n \; M \; G \tag{14}$$

Since both E_n and $M\ G$ are integer matrices the initial difference vector and consequently the forward differences (computed by Equation 8) are all integers. The variables of the algorithm are also scaled up by $2n^3$, and thus the use of the step ϵ and the floating-point variables within the loop of the algorithm are completely avoided. In particular, the parameter t is now an integer, stepping by one from 0 to n. Note that the constants n^3 and $2n^3$ should be computed only once at initialization.

The algorithm *FAST_CURVE*, displayed in Figure 2, is the algorithm *CURVE* modified to include the all-integer decision processes. The time complexity of this efficient version of the algorithm is also linear with the number of voxels written into the CFB. However, writting a single voxel requires now between 9 to 12 additions, 4 to 7 tests, an increment, all in integer, and 0 to 3 *UPDATE_POS* calls and one *PUT_VOXEL*. Note also that 0 to 3 calls to *UPDATE_POS* and one *PUT_VOXEL* are faster than a single *WRITE_VOXEL* (see Section 3). This cost per voxel is much more attractive than the original all floating-point loop (cf. Section 5).

```
Find maximum absolute value of dx/dt, dy/dt, dz/dt for
0 ≤ t ≤ 1 (Eqs. 10-12), set n to maximum of these maxima;
Calculate initial difference vectors Δx, Δy, Δz by Eq. 14;
Set Δ⁰x, Δ⁰y, Δ⁰z to 0;
NEW_POS (g[0][0], g[1][0], g[2][0]);          "start point"

for (t = 0; t ≤ n; t++) {                     "follow the curve"
    if (Δ⁰x > n³) {                           "decision for x"
        UPDATE_POS (X,1);                     "increment x"
        Δ⁰x -= 2n³; }
    else if (Δ⁰x < -n³) {
        UPDATE_POS (X,-1);                    "decrement x"
        Δ⁰x += 2n³; }
    Δ⁰x += Δ¹x;
    Δ¹x += Δ²x;
    Δ²x += Δ³x;

    if (Δ⁰y > n³) {                           "decision for y"
        UPDATE_POS (Y,1);                     "increment y"
        Δ⁰y -= 2n³; }
    else if (Δ⁰y < -n³) {
        UPDATE_POS (Y,-1);                    "decrement y"
        Δ⁰y += 2n³; }
    Δ⁰y += Δ¹y;
    Δ¹y += Δ²y;
    Δ²y += Δ³y;

    if (Δ⁰z > n³) {                           "decision for z"
        UPDATE_POS (Z,1);                     "increment z"
        Δ⁰z -= 2n³; }
    else if (Δ⁰z < -n³) {
        UPDATE_POS (Z,-1);                    "decrement z"
        Δ⁰z += 2n³; }
    Δ⁰z += Δ¹z;
    Δ¹z += Δ²z;
    Δ²z += Δ³z;

    PUT_VOXEL ();                             "in CFB"
}
```

Figure 2: *FAST_CURVE* - An Efficient Algorithm for 3D Scan-Converting Curves

Both algorithms *CURVE* and *FAST_CURVE* handle the three coordinates independently, and therefore one, two and/or three coordinates may be simultaneously changed at any step. Consequently, the resulting curve is a 26-connected path in Z^3 of length $n_{26}=\max(n_x, n_y, n_z)$ voxels. In order to obtain a 6-connected path, that is, one coordinate at the most changes at any step, the algorithm may increment (decrement) only the coordinate with the largest magnitude decision variable. For example, if $|\Delta^0 x|$ is the largest, then x is the only candidate for a change, which is contingent to $|\Delta^0 x| > n^3$. The length in voxels of this curve is $n_6 = n_x + n_y + n_z$.

Similarly, in order to obtain an 18-connected path, i.e., two coordinates at the most change at any step, the algorithm does not change the coordinate with the smallest magnitude of the decision variable. The other two coordinates, however, may or may not be incremented (decremented), depending on whether their respective decision variable magnitude is greater than n^3. The length in voxels of this curve is $n_{18} = \max(n_{26}, \lceil n_6/2 \rceil)$.

7. Parametric Polynomial Surfaces

Bicubic parametric polynomial surfaces are cubic polynomials of two parameters t and u. As both parameters vary across the (0, 1) range, all the points on the surface patch are defined. The two-parameter representation is:

$$f(t,u) = T\ M\ G\ M^{\dagger}\ U^{\dagger} \qquad 0 \le t, u \le 1 \tag{15}$$

$$U = [u^3\ \ u^2\ \ u\ \ 1] \tag{16}$$

T and M are the same as for curves. Again, we use the Bezier form where G is the 4×4 control point matrix, and G_{11}, G_{14}, G_{41}, G_{44} are the surface patch "corners". The properties of the Bezier curves also hold for the Bezier surfaces. Therefore, the forward difference method can be used for surfaces too. The initial 4×4 forward difference matrix Δf_{00} is obtained by multiplying the difference operator by the algebraic matrix $M\ G\ M^{\dagger}$ [9]:

$$\Delta f_{00} = E_{\epsilon}\ M\ G\ M^{\dagger}\ E_{\delta}^{\dagger} \tag{17}$$

where ϵ and δ are the step size in the t and the u parameters, respectively. A bicubic surface in t and u can be drawn as a sequence of cubic curves, where u is a constant and t varies from 0 to 1 using a curve forward difference process. The inter-curve iteration is very similar to that of the intra-curve, but it is performed on all rows of Δf_{00}. This is further explained in the next section in the context of the 3D scan-conversion algorithm for surfaces.

8. 3D Scan-Conversion of Surfaces

The formal data type **surface** represents a bicubic surface patch:

```
typedef struct surface {
    int g[3][4][4];          "x, y, z for 16 control points"
    color c;                 "surface color"
} surface;
```

The 3×4×4 control matrix g is the 16 Bezier geometric control points of the bicubic Bezier patch. When scan-converting such a surface we guarantee lack of 6-connected tunnels. This is achieved if the first difference (in first approximation - the partial derivative) of the surface in each of the parameters is bounded by 1 in magnitude. Thus, we must find the extrema of the derivatives over the surface patch, n for t and m for u, and set the step size in each parameter to their inverse. These extrema are difficult to compute, so we shall use the maxima of the derivatives of the components, and require that they be less than $1/\sqrt{3}$ to guarantee the above requirement. Unfortunately, even finding these extrema is not trivial and requires solving high-degree equations which is very time consuming, and therefore not acceptable.

Instead, we will show that the derivatives of a Bezier surface are also Bezier surfaces, and the convex hull property guarantees bounding derivatives. Finding the bound on $\partial f/\partial t$ is as follows. The D_t-differentiating operator and the derivative of the surface with respect to the parameter t are:

† means transposed

$$D_t = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 3 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \tag{18}$$

$$\frac{\partial f\,(t,u)}{\partial t} = T\ D_t\ M_b\ G\ M_b^{\dagger}\ U^{\dagger} \tag{19}$$

$$= T\ M_b\ (M_b^{-1}\ D_t\ M_b\ G)\ M_b^{\dagger}\ U^{\dagger}$$

The term in parentheses in Equation 19 is the control point matrix G_f' for the $\partial f\ /\partial t$ surface in Bezier representation, thus its D_{t_b}-differentiating operator in Bezier representation is:

$$D_{t_b} = M_b^{-1}\ D_t\ M_b = \begin{bmatrix} -3 & 3 & 0 & 0 \\ -1 & -1 & 2 & 0 \\ 0 & -2 & 1 & 1 \\ 0 & 0 & -3 & 3 \end{bmatrix} \tag{20}$$

Now we find the bound on the derivative as the maximum of the offset between all the control points of the $\partial f\ /\partial t$ surface. After finding all the offsets, set:

$$\begin{aligned} n_x' &= \max(G_x')\text{-}\min(G_x') \\ n_y' &= \max(G_y')\text{-}\min(G_y') \\ n_z' &= \max(G_z')\text{-}\min(G_z') \end{aligned} \tag{21}$$

$$\epsilon = \frac{1}{n} = \frac{1}{\sqrt{3}} \min\left(\frac{1}{n_x'}, \frac{1}{n_y'}, \frac{1}{n_z'}\right)$$

Similarly, calculate δ and m using the D_u matrices. Note that symmetry provides $D_u = D_t^{\dagger}$.

The algorithm to scan-convert a 3D bicubic parametric surface employs the forward differences iteration to evaluate the surface as a sequence of cubic curves $f\,(t,0)$, $f\,(t,\delta)$, $f\,(t,2\delta)$, $\cdots$, each of which is computed using a 1D curve forward difference iteration similar to the one described for curves. The computation of the initial endpoint of each of the curves is done by a similar 2D iteration.

The algorithm first calculates the difference matrix Δf_{00} (using Equation 17) for each of its components, i.e., the 4×4 matrices Δx_{tu}, Δy_{tu}, and Δz_{tu}. The zero column of Δx_{tu} contains the initial values $\Delta^0 x_t$, $\Delta^1 x_t$, $\Delta^2 x_t$, $\Delta^3 x_t$ used as the x forward differences of the curve $f\,(t,u_0)$ for the current u_0 (u_0 is constant, t varies). Similarly the zero column of Δy_{tu} and Δz_{tu} are utilized for the y and z components. After scan-converting the curve $f\,(t,u_0)$ the difference matrices Δx_{tu}, Δy_{tu}, and Δz_{tu} are updated as 2D forward differences. Namely, column 1 of these matrices is added to their column 0, column 2 is added to column 1, and column 3 is added to column 2. Now column 0 contains the initial difference vectors for the next curve $f\,(t,u_0+1)$.

The algorithm complexity is $\mathbf{O}(n\ m)$ which is linear in the number of painted voxels. Each voxel drawn in the inner loop of such an algorithm requires the calculation of 10 non-integer additions, a voxel write (including 3 *ROUND* operations), and an inner loop completion test. This is in addition to 37 non-integer additions, copying 12 elements, and a loop completion test for each u. Initialization involves the non-integer computation of the coefficients, ϵ, δ, and the initial difference matrices.

9. Efficient 3D Scan-Conversion of Surfaces

Like for the curve algorithm, the heavy non-integer computations can be replaced by a more efficient all-integer algorithm, *FAST_SURFACE*, presented in Figure 3. The idea is very similar to that used by the *FAST_CURVE* algorithm, namely, scaling the program variables up by a large constant so that all the algorithm variables, including the forward differences, are integers, and using decision processes to set the coordinates of the next voxel. More specifically, the matrix E_δ is redefined as E_m:

$$E_m = m^3\ E_\delta = \begin{bmatrix} 0 & 0 & 0 & m^3 \\ 1 & m & m^2 & 0 \\ 6 & 2m & 0 & 0 \\ 6 & 0 & 0 & 0 \end{bmatrix} \tag{22}$$

while the matrix E_ϵ is redefined as E_n (Equation 13). As a result the initial difference matrix Δf_{00}, defined originally in Equation 17, is redefined as:

$$\Delta f_{00} = E_n\ P\ E_m^{\dagger} = 2n^3 m^3 E_\epsilon\ P\ E_\delta^{\dagger} \tag{23}$$

The coordinates of the next voxel along the curve $f\,(t,u_0)$ are selected based on three decision processes, one for each dimension. In the x decision process, for example, the absolute magnitude of the decision variable $\Delta^0 x_t$ is tested against the threshold $n^3 m^3$ to determine whether to increment, decrement or leave unchanged the X register holding the x ordinate. These inner loop decision processes are identical to those employed by the *FAST_CURVE* algorithm.

The curve scan-conversion process is repeated for all the $m+1$ curves, $u = 0,1,\ldots,m$. In order to get the initial difference values for the next curve, the algorithm updates the difference matrices as a 2D forward difference and then steps along the curve $f\,(0,u)$ (see Figure 3). The scan-conversion of this curve also involves three decision processes. The decision process for x, for example, tests the decision variable $\Delta^{00} x_{tu}$, and the variable x_u is accordingly incremented, decremented, or remains unchanged. The point (x_u, y_u, z_u) is the starting point of the next curve to be scan-converted by the inner loop. Note, that unlike the variables Δx_{tu}, Δy_{tu}, Δz_{tu} which are scaled up by the scalar $2n^3 m^3$, the coordinates (x_u, y_u, z_u), starting at the first endpoint, represents all along actual voxel coordinates.

The time complexity of the efficient version of the algorithm is also linear in the number of voxels written into the CFB. However, writting one voxel in the inner loop requires the same number of integer operations as for *FAST_CURVE*, namely, 9-12 additions, 4-7 tests, an increment, 0-3 *UPDATE_POS* calls, and one *PUT_VOXEL*. The outer loop requires the following integer operations for each u: 36-39 additions, 1-4 increments, 4-7 tests, 15 copies, and one *NEW_POS* call. This is much more attractive than the original surface scan-conversion algorithm (cf. Section 8).

10. Parametric Polynomial Volumes

Parametric polynomial curves and surfaces have their trivariate version, i.e, a volume element. Varying the three parameters t, u, v from 0 to 1 defines all the points of the volume element. The extension to volumes is similar to the way curves have been extended into surfaces. Namely, if one parameter, say v, is assigned a constant value, and the other

```
Find n and m as described by Eq. 21;
Find initial difference matrices Δx_tu , Δy_tu , Δz_tu using Eq. 17;
Set Δ^00 x_tu , Δ^00 y_tu , Δ^00 z_tu to 0;
x_u = g[0][0][0];
y_u = g[1][0][0];
z_u = g[2][0][0];
for (u = 0; u ≤ m; u++) {
        NEW_POS(x_u ,y_u ,z_u );                    "start point"
        Copy column 0 of Δx_tu to Δx_t ;    Set Δ^0 x_t to 0;
        Copy column 0 of Δy_tu to Δy_t ;    Set Δ^0 x_t to 0;
        Copy column 0 of Δz_tu to Δz_t ;    Set Δ^0 x_t to 0;
        for (t = 0; t ≤ n; t++) {                   "follow f(*,u)"
                if (Δ^0 x_t > n^3 m^3) {
                        UPDATE_POS (X,1);
                        Δ^0 x_t -= 2n^3 m^3;
                }
                else if (Δ^0 x_t < -n^3 m^3) {
                        UPDATE_POS (X,-1);
                        Δ^0 x_t += 2n^3 m^3;
                }
                Δ^0 x_t += Δ^1 x_t ;
                Δ^1 x_t += Δ^2 x_t ;
                Δ^2 x_t += Δ^3 x_t ;
                Same decisions for Δy_t and Δz_t ;
                PUT_VOXEL ();
        }
        Update Δx_tu : col 0 += col 1;
                       col 1 += col 2;
                       col 2 += col 3;
        Same update for Δy_tu and Δz_tu ;
        if (Δ^00 x_tu > n^3 m^3) {                  "decision for f(0,*)"
                x_u ++;
                Δ^00 x_tu -= 2n^3 m^3;
        }
        else if (Δ^00 x_tu < -n^3 m^3) {
                x_u --;
                Δ^00 x_tu += 2n^3 m^3;
        }
        Same decisions for Δy_tu and Δz_tu ;
}
```

Figure 3: *FAST_SURFACE* - An Efficient Algorithm for 3D Scan-Converting Surfaces

two parameters, t and u, are varied in the range 0 to 1, a bivariate surface is defined. As v varies from 0 to 1 a sequence of bivariate surfaces sweeps the entire volume element.

Since the geometric matrix for volume representations is a tensor of 3-component vectors, the following summation notation is used:

$$f(t,u,v) = \sum_{i=0}^{3} \sum_{j=0}^{3} \sum_{k=0}^{3} h_i(t)\, h_j(u)\, h_k(v)\, g_{ijk} \qquad (24)$$

where $h_i(t)$, $h_j(u)$, and $h_k(v)$ are the geometric basis functions, and g_{ijk} are the components of the geometric tensor. The properties of the Bezier curves and surfaces hold also for the Bezier type volumes. More specifically, the forward difference method can be generalized to volumes.

11. 3D Scan-Conversion of Volumes

The trivariate Bezier volume is formally represented by the data type **volume**:

```
typedef struct volume {
        int g[3][4][4][4];      "x, y, z for 64 control points"
        color c;                "volume color"
} volume;
```

The 3×4×4×4 control matrix g is the 64 Bezier geometric control points of a tricubic Bezier volume element in the parameters t, u, and v, each varies from 0 to 1. A volume element is scan-converted as a sequence of bicubic surfaces, where v is constant and t and u vary. Consequently, the scan-conversion algorithm for tricubic volumes is an extension of the associated algorithm for bicubic surfaces.

Furthermore, the all-integer version of the surface algorithm, *FAST_SURFACE*, can be generalized to volumes. An efficient volume algorithm, *FAST_VOLUME*, employing only integer arithmetic is displayed in Figure 4. The complexity of the algorithm is $\mathbf{O}(n\ m\ l)$, which is linear in the number of painted voxels. The numbers n, m and l are the inverse of the step sizes in t, u and v, respectively, and their computation is similar to that for surfaces and 6-connected curves. In order to avoid internal cavities in the solid volume, the algorithm generates a 6-connected volume, i.e., at each step of the algorithm only one coordinate may be changed. This coordinate is the one with the largest magnitude decision variable.

12. Concluding Remarks

We have described 3D scan-conversion algorithms for Bezier parametric cubic curves, bicubic surfaces, and tricubic volumes, from their geometric R^3 representation to the Z^3 voxel-image space. The conversion was achieved while obeying the fidelity, connectivity and efficiency requirements. For the curve algorithms 6-, 18- and 26-connected versions have been discussed. The surface algorithm guarantees lack of 6-connected tunnels in the converted surface, while the volume element generated by the volume algorithm has no internal cavities. All the algorithms are incremental and use only simple operations within their inner loops. Furthermore, efficient integer-only algorithms for 3D scan-conversion of curves, surfaces and volumes have been developed.

All the algorithms do scan-conversion with computational and temporal complexities which are linear in the number of voxels in the object. An object given for scan-conversion is assumed to be completely within the CFB bounds. However, if it extends slightly outside the CFB, scissoring can be applied, which can be easily incorporated within all the algorithms with no added time complexity.

All the 3D scan-conversion algorithms were implemented as part of the 3D geometry processor of the CUBE architecture. The geometry processor has been simulated in software, written in C under UNIX running on VAX computers and SUN workstations. Special purpose hardware (third order DDAs) has also been designed to improve conversion speed. Figure 5, generated in less than 30 seconds on a color SUN 3/160C, shows a straw dipped in a semi-transparent goblet filled with semi-transparent wine. The goblet was scan-converted by the geometry processor into a 128^3 CFB using the efficient surface algorithm. The image was then projected and rendered (with semi-transparency and shading) by the viewing processor of

```
Find n, m, l;  Find initial matrices Δx_tuv , Δy_tuv , Δz_tuv ;
Set Δ^000 x_tuv , Δ^000 y_tuv , Δ^000 z_tuv  to 0;
x_v = g[0][0][0][0];  y_v = g[1][0][0][0];  z_v = g[2][0][0][0];
for (v = 0;  v ≤ l;  v++) {
        Δx_tu  = 0 t-u layer of Δx_tuv ;
        Δy_tu  = 0 t-u layer of Δy_tuv ;
        Δz_tu  = 0 t-u layer of Δz_tuv ;
        Set Δ^00 x_tu , Δ^00 y_tu , Δ^00 z_tu  to 0;
        x_u = x_v ;   y_u = y_v ;   z_u = z_v ;
        for (u = 0;  u ≤ m;  u++) {          "surface f(*,*,v)"
                NEW_POS(x_u , y_u , z_u );
                Δx_t  = 0 col of Δx_tu ;
                Δy_t  = 0 col of Δy_tu ;
                Δz_t  = 0 col of Δz_tu ;
                Set Δ^0 x_t , Δ^0 y_t , Δ^0 z_t  to 0;
                for (t = 0;  t ≤ n;  t++) {  "curve f(*,u,v)"
                        if ( | Δ^0 x_t  | is the largest) {
                                if (Δ^0 x_t  > n^3 m^3 l^3) {
                                        UPDATE_POS (X,1);
                                        Δ^0 x_t  -= 2n^3 m^3 l^3;  }
                                else if (Δ^0 x_t  < -n^3 m^3 l^3) {
                                        UPDATE_POS (X,-1);
                                        Δ^0 x_t  += 2n^3 m^3 l^3;  }
                                Δ^0 x_t  += Δ^1 x_t ;
                                Δ^1 x_t  += Δ^2 x_t ;
                                Δ^2 x_t  += Δ^3 x_t ;
                        }                          "step only in x"
                        else if ( | Δ^0 y_t  | is the largest)
                                Same decisions for Δy_t ;
                        else    Same decisions Δz_t ;
                        PUT_VOXEL ();
                }
                Update Δx_tu ,Δy_tu ,Δz_tu :    col 0 += col 1;
                                                col 1 += col 2;
                                                col 2 += col 3;
                if ( | Δ^00 x_tu  | is the largest) {
                        if (Δ^00 x_tu  > n^3 m^3 l^3) {  x_u ++;
                                Δ^00 x_tu  -= 2n^3 m^3 l^3;  }
                        else if (Δ^00 x_tu  < -n^3 m^3 l^3) {  x_u --;
                                Δ^00 x_tu  += 2n^3 m^3 l^3;  }
                }                                  "step only in x"
                else if ( | Δ^00 y_tu  | is the largest)
                        Same decisions for Δy_tu ;
                else    Same decisions for Δz_tu ;
        }
        Update Δx_tuv ,Δy_tuv ,Δz_tuv :  layer 0 += layer 1;
                                         layer 1 += layer 2;
                                         layer 2 += layer 3;
        if ( | Δ^000 x_tuv  | is the largest) {
                if (Δ^000 x_tuv  > n^3 m^3 l^3) {  x_v ++;
                        Δ^000 x_tuv  -= 2n^3 m^3 l^3;  }
                else if (Δ^000 x_tuv  < -n^3 m^3 l^3) {  x_v --;
                        Δ^000 x_tuv  += 2n^3 m^3 l^3;  }
        }                                          "step only in x"
        else if ( | Δ^000 y_tuv  | is the largest)
                Same decisions for Δy_tuv ;
        else    Same decisions for Δz_tuv ;
}
```

Figure 4: *FAST_VOLUME* - An Efficient Algorithm for 3D Scan-Converting Volumes

the CUBE architecture.

The algorithms developed here are by no means limited to cubic Bezier objects and apply equally well to other forms, and can also be extended to higher degree objects. Furthermore, the 3D scan-conversion algorithms for curves, surfaces and volumes, and more specifically the efficient all-integer decision mechanisms can also be exploited in conventional pixel-based algorithms (cf. [4]) which generate directly (after eliminating hidden surfaces) a set of pixels representing a projection of the desired parametric object.

Acknowledgment

This work was supported by the National Science Foundation under grant DCR-86-03603. The author would like to thank Eyal Shimony of the Center of Computer Graphics, Ben-Gurion University, for his work on an early version of some of the algorithms.

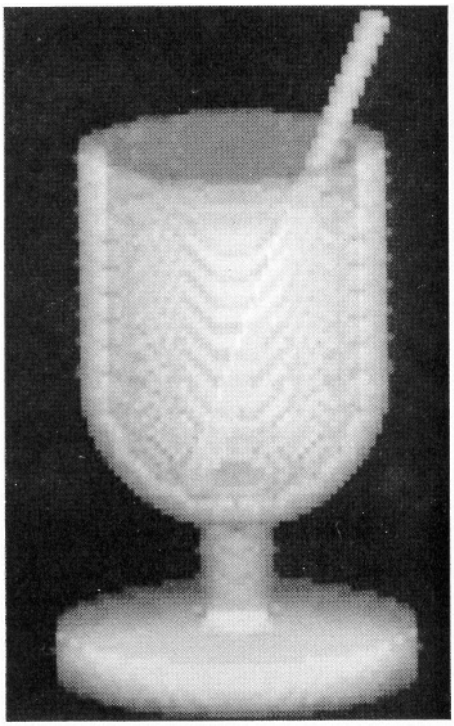

Figure 5: A Goblet Scan-Converted into a 128^3 CFB Using the Surface Algorithm and Projected/Rendered with Semi-Transparencies and Shading

13. References

1. Bezier, P., "UNISURF System: Principles, Program, Language", in *Computer Languages for Numerical Control*, J. Hatvany, (ed.), North-Holland, Amsterdam-London, 1972, 417-426.
2. Bezier, P., *Numerical Control - Mathematics and Applications*, A. R. Forrest (Trans.), Wiley, London, 1972.
3. Bezier, P., "Mathematical and Practical Possibilities of UNISURF", in *Computer Aided Geometric Design*, R. E. Barnhill and R. F. Riesenfeld, (eds.), Academic, New York, 1974, 127-152.
4. Blinn, J. F., Carpenter, L., Lane, J. and Whitted, T., "Scan Line Methods for Displaying Parametrically Defined Surfaces", *Communications of the ACM*, **23**, 1 (January 1980), 23-34.
5. Bresenham, J. E., "Algorithm for Computer Control of a Digital Plotter", *IBM Systems Journal*, **4**, 1 (1965), 25-30.
6. Clark, J. H., "Parametric Curves, Surfaces, and Volumes in Computer Graphics and Computer-Aided Geometric Design", Technical Report 221, Computer Systems Laboratory, Stanford University, Stanford, CA, November 1981.
7. Coons, S. A., "Surfaces for Computer-Aided Design of Space Forms", MIT Project MAC Tech. Rep.-41, June 1967.
8. deBoor, C., "On Uniform Approximation with Splines", *Journal of Approximation Theory*, **1**, (1968), 249-274.
9. Foley, J. D. and van Dam, A., *Fundamentals of Interactive Computer Graphics*, Addison-Wesley, Reading, MA, 1982.
10. Forrest, A. R., "On Coons and Other Methods for the Representation of Curved Surfaces", *Computer Graphics and Image Processing*, **1**, 4 (December 1972), 341-354.
11. Goldwasser, S. M., "A Generalized Object Display Processor Architecture", *IEEE Computer Graphics and Applications*, **4**, 10 (October 1984), 43-55.
12. Jackel, D., "The Graphics PARCUM System: A 3D Memory Based Computer Architecture for Processing and Display of Solid Models", *Computer Graphics Forum*, 1985, 21-32.
13. Kaufman, A. and Bakalash, R., "CUBE - An Architecture Based on a 3-D Voxel Map", in *Theoretical Foundations of Computer Graphics and CAD*, R. A. Earnshaw, (ed.), Springer-Verlag, 1987.
14. Kaufman, A. and Bakalash, R., "A 3-D Cellular Frame Buffer", *Proc. EUROGRAPHICS'85*, Nice, France, September 1985, 215-220.
15. Kaufman, A. and Bakalash, R., "Memory and Processing Architecture for 3-D Voxel-Based Imagery", Technical Report 87/06, Department of Computer Science, SUNY at Stony Brook, February 1987.
16. Kaufman, A., "An Algorithm for 3D Scan-Conversion of Polygons", *Proc. EUROGRAPHICS'87*, Amsterdam, Netherlands, August 1987.
17. Kaufman, A., "Voxel-Based Architectures for Three-Dimensional Graphics", *Proc. IFIP'86*, Dublin, Ireland, September 1986, 361-366.
18. Kaufman, A. and Shimony, E., "3D Scan-Conversion Algorithms for Voxel-Based Graphics", *Proc. ACM Workshop on Interactive 3D Graphics*, Chapel Hill, NC, October 1986.
19. Kim, C. E., "Three-Dimensional Digital Planes", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, **PAMI-6**, 5 (September 1984), 639-645.
20. Lane, J. M. and Riesenfeld, R. F., "A Theoretical Development for the Computer Generation and Display of Piecewise Polynomial Surfaces", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, **PAMI-2**, 1 (January 1980), 35-46.
21. Ohashi, T., Uchiki, T. and Tokoro, M., "A Three-Dimensional Shaded Display Method for Voxel-Based Representation", *Proc. EUROGRAPHICS'85*, Nice, France, September 1985, 221-232.
22. Pavlidis, T., *Algorithms for Graphics and Image Processing*, Computer Science Press, Rockville, MD, 1982.
23. Riesenfeld, R. F., "Applications of B-spline Approximation to Geometric Problems of Computer Aided Design", University of Utah UTEC-CSc-73-126, March 1973.
24. Rosenfeld, A., "Three-Dimensional Digital Topology", Computer Science Center, Univ. of Maryland, Tech. Rep.-936, 1980.
25. Srihari, S. N., "Representation of Three-Dimensional Digital Images", *Computing Surveys*, **4**, 13 (December 1981), 399-424.