# 3D Scan-Conversion Algorithms for Voxel-Based Graphics

**Arie Kaufman**

Department of Computer Science
SUNY at Stony Brook
Stony Brook, NY 11794-4400, USA
ari@suny-sb

and

**Eyal Shimony**

Center of Computer Graphics
Department of Mathematics & Computer Science
Ben-Gurion University of the Negev
Beer-Sheva 84120, Israel

***Abstract***

An assortment of algorithms, termed *three-dimensional (3D) scan-conversion algorithms*, is presented. These algorithms scan-convert 3D geometric objects into their discrete voxel-map representation within a *Cubic Frame Buffer (CFB).* The geometric objects that are studied here include three-dimensional lines, polygons (optionally filled), polyhedra (optionally filled), cubic parametric curves, bicubic parametric surface patches, circles (optionally filled), and quadratic objects (optionally filled) like those used in constructive solid geometry: cylinders, cones, and spheres.

All algorithms presented here do scan-conversion with computational complexity which is linear in the number of voxels written to the CFB. All algorithms are incremental and use only additions, subtractions, tests and simpler operations inside the inner algorithm loops. Since the algorithms are basically sequential, the temporal complexity is also linear. However, the polyhedron-fill and sphere-fill algorithms have less than linear temporal complexity, as they use a mechanism for writing a voxel run into the CFB. The temporal complexity would then be linear with the number of pixels in the object's 2D projection. All algorithms have been implemented as part of the *CUBE Architecture*, which is a voxel-based system for 3D graphics. The CUBE architecture is also presented.

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

## 1. Introduction

Graphical data are commonly represented in one of three primary forms: as a *continuous object (geometric) model*, as a *discrete pixel-image raster*, and as a *discrete voxel-image space*. The former two representations have received considerable attention in research and development, and in the literature. There is an abundance of scan-conversion algorithms, that scan-convert 2D continuous geometric representations into sets of pixels in the pixel-image plane. We focus on algorithms, termed *three-dimensional (3D) scan-conversion algorithms*, that fall in another category. Such an algorithm converts a 3D continuous representation into a set of voxels in the discrete voxel-image space.

A voxel-image space system, in which the 3D scan-conversion algorithms have been tested, is the *CUBE Architecture* [19, 20]. The theme of this system is based on the concept that the 3D inherently continuous scene is discretized, sampled, and stored in a large 3D *Cubic Frame Buffer (CFB)* of unit cells called volume elements or *voxels*. Three-dimensional objects are digitized, and a regularly spaced array of values is obtained and stored in the CFB. The sampling yields a cellular tessellation of the original volume. Such a cellular cubic memory provides a real, yet discrete model of reality. A CFB with 512-cube resolution and 8-bit-deep voxels, for example, is a large memory buffer of size 128M bytes. Since computer memories are significantly decreasing in price and increasing in their compactness, such huge memories are becoming more and more feasible.

One of the processors of the CUBE architecture, the *3D Geometry Processor*, has access to the CFB to generate CFB images. It scan-converts 3D geometric objects into sets of voxels within the CFB approximating the original objects. It is basically an algorithmic-oriented processor, where for each geometric object a 3D scan-conversion algorithm, like those presented in this paper, is applied.

A large body of algorithms to scan-convert 2D geometric objects, which are fundamental to 2D raster systems, has been published. The objects that have been studied are: straight lines [4, 32, 34, 35], curves [9, 18, 30, 35], and more specifically circles and circular arcs [6, 16, 23, 34], ellipses and parabolae [28, 36]. Scan-conversion algorithms for planar polygons [12, 24, 26], and planar regions bounded by curves [15, 27, 37], have also been investigated. This paper presents an assortment of 3D scan-conversion algorithms, few of which are generalizations of the common 2D algorithms into the 3D case. However, most of the algorithms are, to the best of our knowledge, original schemes. The output of these algorithms are "colored" discrete points (voxels), or runs of voxels parallel to a major axis ($x$ , $y$ or $z$ ), written to the CFB.

Scan-conversion algorithms for the following 3D geometric objects are described and analyzed:

- Straight line segments
- Polygons (optionally filled)
- Polyhedra (optionally filled)
- Cubic parametric curves
- Bicubic parametric surface patches
- Circles (optionally filled)
- Quadratic objects, such as cylinders, cones, and spheres (optionally filled)

An introduction to the CUBE architecture is given in Section 2, and the auxiliary functions for CFB access are described in Section 3. Section 4 explains terms used in the rest of the article, and Section 5 outlines the assumptions and requirements from a 3D scan-converter. Each of Sections 6-12 presents scan-conversion algorithms used in the 2D case (if any), introduces the 3D object description, the data structures, and the algorithm designed for one of the 3D objects. Descriptions are in either pseudo-C (where the keywords are in **bold** face) and/or in algorithm steps. 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.

## 2. The CUBE Architecture

The *CUBE (CUbic frame BuffEr) Architecture* is centered around a large *Cubic Frame Buffer (CFB)* of voxels. A voxel may assume a numerical value of density in the broad sense, representing the material, color, texture, and translucency ratio of a small unit cube of the real scene. There are three processors, the *3D Geometry Processor*, the *3D Frame-Buffer Processor*, and the *3D Viewing Processor*, which access the CFB, to generate, to manipulate, and to view the CFB images, respectively. The *3D Geometry Processor (GP3)* accepts geometric representations of 3D objects and scan-converts them into their 3D discrete representation within the CFB. The objects handled by the processor are the same 3D objects being studied in this paper. The GP3 executes a collection of algorithms, where for each given geometric object a 3D scan-conversion algorithm is applied. By enabling read-modify-write operations, the GP3 is also capable of performing boolean operations on objects, like those used in constructive solid geometry.

The *3D Frame-Buffer Processor (FBP3)* manipulates and processes the 3D discrete images stored in the CFB. FBP3 is actually a voxel-map engine (an extended version of a 2D *bitblt* [14]) performing such operations as 3D sub-cube manipulation: transformations, copying, loading, storing, painting, voxelwise operations, etc. It also acts as an interface for 3D input devices and as an input channel for 3D scanned data. The primitives of the FBP3 are 3D sub-arrays of the CFB of three types: 3D *boxes*, 3D *jacks* and 3D *figurines*. A box is

rectangular (i.e., a cuboid, a 3D window) defined by two opposite corners and a priority value which is a unique value for each box facilitating the support of overlapping boxes. Jacks (i.e., 3D cursors) provide a flexible feedback mechanism, assist the user in touring the interior of the 3D objects and improve user's orientation in the 3D environment. Figurines (i.e., 3D icons, *vicons*) are small application-oriented 3D constant images, that are either bounded to a box or may float independently.

The projections of the CFB images onto the 2D graphics display are accomplished by a *3D Viewing Processor (VP3)*. The VP3 generates from the CFB parallel orthographic projections and arbitrary parallel projections. In order to handle the huge throughput a special common bus has been designed. It selects the opaque voxel closest to the assumed observer in a given beam, in a time which is proportional to the *log* of the length of the beam. The common bus concept is also appropriate for yon and hither clipping, for selecting isochromatic surfaces, and for handling semi-transparent surfaces. The VP3 also renders the 2D image during the projection process, implementing semi-transparency, lighting and shading, and pseudo-colors.

A unique memory organization [22], which fetches/stores a full beam of voxels simultaneously, enables the processors to handle beams rather than single voxels, and thus enabling the system to cope with real-time constraints. In order to fetch/store a full beam of voxels simultaneously, a special 3D symmetric organization of the $n^3$ CFB has been designed. It consists of diagonal parallel sections having a 45 degree angle with the main axes planes, and are grouped in $n$ distinct memory modules. A voxel with space coordinates $(x, y, z)$ is being mapped onto the $k$-th module by:

$$k = (x + y + z) \ mod \ n \tag{1}$$

The internal mapping $(i, j)$ within the memory module is given by:

$$i = x, \qquad j = y \tag{2}$$

Since two coordinates are always constant along any beam, regardless of the view direction, the third coordinate guarantees that one and only one voxel from the beam resides in any one of the modules.

The CUBE architecture and a general-purpose real-time *3D CUBE Workstation*, which is based on it, are appropriate for many 3D applications, such as: medical imaging, computer aided design, 3D animation and simulation, 3D image processing and pattern recognition, quantitative microscopy, and general 3D graphics interaction. 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 (e.g., a synthetic injection needle superimposed on an ultrasound image.)

There are three other voxel-image space systems, GODPA [13], PARCUM [17],

and 3DP$^4$ [25], which are all viewing engines that assume the existence of a database of voxel-based objects for loading and image viewing. Unlike these systems that operate solely in the discrete voxel-image space, CUBE further provides object space capabilities and ways to intermix the two (see also [21]). These capabilities are enabled using a spectrum of 3D scan-conversion algorithms, which are introduced in this paper. These algorithms enable the CUBE system to cater for a wider variety of applications which accept a geometric representation of the 3D scene, as well as those that generate synthetic images compiled from sampled experimental data, a voxel-based database, and the geometric model.

## 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:

1) *NEW_POS (axis, value);*
Where *axis* is one of $X$, $Y$ or $Z$, and *value* is the ordinate value. This function executes in one machine instruction and should therefore be fast.

2) *NEW_COLOR (c);*
Updates the *COLOR* register with the color $c$.

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.

Using these functions we define voxel write macros:

```
1) VOXEL_POS (x, y, z) {
       NEW_POS (X, x);
       NEW_POS (Y, y);
       NEW_POS (Z, z);
   }

2) WRITE_VOXEL (x, y, z, c) {
       VOXEL_POS (x, y, z);
       NEW_COLOR (c);
       PUT_VOXEL ();
   }
```

The following functions are available for writing to the CFB a run of uniform color voxels that is parallel to one of the axes:

1) *LOAD_MIN_VALUE (value);*

2) *LOAD_MAX_VALUE (value);*

3) *WRITE_RUN (axis);*

Where *LOAD_MIN_VALUE* and *LOAD_MAX_VALUE* load a boundary *value* to the *MIN_VALUE* and *MAX_VALUE* registers, respectively. *WRITE_RUN* writes the voxel run (beam) from *MIN_VALUE* to *MAX_VALUE* along the *axis* , using values in the $X$ , $Y$ , or $Z$ registers for the other two dimensions.

## 4. Terminology

The 3D discrete topology terms used here are generalizations of those used in 2D discrete topology. Most of the terms are in accordance with those presented in [26, 31, 33]. Let the continuous 3D space $(R \times R \times R)$ be marked as $R^3$, while the discrete 3D voxel-image space $(Z \times Z \times Z)$, which is a 3D array of grid points, shall be referred to as $Z^3$. 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 in this paper.

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

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

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

In $Z^3$ we further define 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.

## 5. Assumptions and Requirements

The following basic assumptions were made about the coordinate space of the geometric objects to be scan-converted. An object given for scan-conversion is assumed to be in final form, i.e. no further spatial transformations are necessary, and it is completely within the CFB bounds, thus avoiding necessity for clipping.

However, if it extends outside the CFB, scissoring can be applied, where voxels outside the CFB are ignored. This is relatively expensive, unless the object extends only slightly outside. Scissoring can be easily incorporated within all the algorithms with no added time complexity. The scissoring operation can also be applied to scan-conversion of 3D objects bounded by planes.

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 requirements assume viewing from a major axis, and would have to be changed slightly for omni directional viewing.

### 5.1. The Fidelity Requirement

An object in the continuous space is a subset of the space $R^3$. The basic fidelity requirements in scan-converting 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 to be converted is a point in $R^3$ (0D), the converted object is the discrete point in which region the $R^3$ point resides.

4) If the object is 1D (line segment or curve), the converted object will meet certain connectivity requirements. The converted endpoints will always be in the converted object.

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

6) If the object is 3D (volume), its "inside" will be converted according to requirements 1 and 2. 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.

### 5.2. The Connectivity Requirement

1) For a 1D object we shall require 6-connectivity or 26-connectivity, depending on implementation needs.

2) For a 2D object we shall require "lack of tunnels" of certain connectivity. Again, 6-connected or 26-connected tunnels can be disallowed depending on

implementation requirements.

Assume that the system requirements are such that we only require viewing along a major axis. Thus, we only require 26-connectivity for 1D objects and disallow only 6-connected tunnels in 2D objects.

### 5.3. The Efficiency Requirement

It seems that in all cases the computational complexity is at most equal to the number of discrete points converted multiplied by a constant. The goals are, thus:

1) Never to suffer a higher complexity.

2) Decrease the constant.

3) Use integer or fixed point arithmetic whenever possible, and floating point only when impossible otherwise.

4) Use simple operations (addition) rather than complex ones (multiplication), and avoid altogether more complex operations (square root) within inner algorithm loops.

5) Operate in parallel so that the temporal complexity is less than the computational complexity, without too much hardware complexity.

## 6. Scan-Converting 3D Lines

Two-dimensional straight line segments are usually defined by their two endpoints, and are scan-converted by one of a variety of algorithms (e.g., BRM - Binary Rate Multiplier, and DDA - Digital Differential Analyzer). The best known algorithm is Bresenham's DDA algorithm [4], which is "optimal", utilizes only integer arithmetic, and converts the line segment with 8-connectivity in $Z^2$. Van Aken and Novak [35] have employed a midpoint method for deriving a line drawing algorithm which does not differ from Bresenham's. Suenaga et al. [34] have proposed a modified displacement comparison method for implicit non-parametric representations of lines. Sproull [32] has developed some parallel multipoint algorithms that generate fixed length sequences of pixels that are written simultaneously into the frame buffer. Other multipoint algorithms, that generate variable run coding related to the repeating patterns in the pixel sequences, have been published [5, 7, 11, 29].

A 3D straight line segment is defined by its two endpoints, which have integer coordinates, and its color:

```
typedef struct line {
        int p1[3];              "First endpoint"
        int p2[3];              "Second endpoint"
        color c;                "Color of line"
} line;
```

The algorithm *LINE* to scan-convert a 3D line draws a 26-connected line between *p1* and *p2*. The algorithm, presented in Figure 1, is actually a double Bresenham's 2D line algorithm. *LINE* takes unit steps along the direction with the largest difference between its endpoint coordinates. Without detracting from the generality of the algorithm, assume that this direction is $x$, and that $x2 > x1$. At each $x$ step, the algorithm determines the $y$-coordinate and the $z$-coordinate of the voxel nearest the desired line path. To determine this, two integer decision variables $dy$ and $dz$ are computed incrementally.

For the sake of brevity, the discussion of the decision procedure will be limited to the case $\Delta x \geq \Delta y, \Delta z \geq 0$. Voxel $V$ at $(x, y, z)$ represents the voxel that had been selected to be the closest to the desired line. At the current step the algorithm decides whether the next voxel closest to the line should be $U=(x+1, y, z)$, $W=(x+1, y+1, z)$, $S=(x+1, y, z+1)$, or $T=(x+1, y+1, z+1)$. The decision is performed in two steps, first for $y$, in the $x-y$ plane, $dy$ is tested. If $dy < 0$ then the line (at $x+1$) is closer to $y$ than to $y+1$, $y$ is not incremented, and the candidates for the next voxel are thus either $U$ or $S$ (depending on $dz$). If, however, $dy \geq 0$, i.e., either $W$ or $T$ is lying closer to the desired line, then $y$ is incremented. The decision for $z$, in the $x-z$ plane, is similar to that of $y$. If $dz < 0$, then $z$ is incremented (either $T$ or $S$ is selected), otherwise, either $W$ or $U$ is selected. The decision variables $dy$ and $dz$ are initialized and incrementally updated for the subsequent step the same way the decision variable is maintained in the 2D Bresenham's algorithm.

The algorithm uses only integer arithmetic, and only addition, subtraction, shifting, and testing are employed. All multiplications by 2 are implemented by left shifts. The inner loop is very simple requiring three additions and three tests for each voxel written into the CFB. In the worst case, where all coordinates change for a given voxel, two additional additions (or two *NEW_POS* calls) are necessary for that voxel.

## 7. Scan-Converting 3D Polygons

A 2D closed polygon is defined by a sequence of line segments (its edges) where the first endpoint of the first edge coincides with the second endpoint of the last edge. A scan-line algorithm for converting a polygon is described in [12, 24, 26]. The algorithm assumes that the polygon edges are sorted along the $y$ axis. It finds the intersections of each scan line with the currently active list of edges,

```
        x = x1;    Δx  = x2 - x1;

"Initialization for y"
        y = y1;    Δy  = ABS (y2 - y1);    ysign = SIGN (y2 - y1);
        dy = 2 * Δy  - Δx ;                "Decision variable along y"
        yinc1 = 2 * Δy ;                   "Increment along y - if dy < 0"
        yinc2 = 2 * (Δy  - Δx );           "Increment along y - if dy ≥ 0"

"Same initialization for z"
        z = z1;    Δz  = ABS (z2 - z1);    zsign = SIGN (z2 - z1);
        dz = 2 * Δz  - Δx ;                "Decision variable along z"
        zinc1 = 2 * Δz ;                   "Increment along z - if dz < 0"
        zinc2 = 2 * (Δz  - Δx );           "Increment along z - if dz ≥ 0"

        WRITE_VOXEL (x, y, z, c);          "First endpoint of line"

"Start following line - algorithm loop"
        while (x < x2)
        {
                x++;                       "Step in x"

                if (dy < 0)                "Then: no change in y"
                        dy += yinc1;       "Update dy"
                else {
                        dy += yinc2;       "Update dy, and"
                        y += ysign;        "Increment/decrement y"
                     }

                if (dz < 0)                "Then: no change in z"
                        dz += zinc1;       "Update dz"
                else {
                        dz += zinc2;       "Update dz, and"
                        z += zsign;        "Increment/decrement z"
                     }

                WRITE_VOXEL (x, y, z, c);  "Next voxel near line"
        }
```

Figure 1: *LINE* - An Algorithm for Scan-Converting 3D Lines

using a DDA and exploiting edge coherency. Edges are removed from or merged with the active list as $y$ increments. The intersections are sorted by increasing $x$-coordinate, and runs of pixels between pairs of intersections are filled in. Pavlidis [27] has presented a similar scheme for scan-converting regions bounded by parabolic splines. Algorithms for scan-converting regions bounded by circular arcs have been proposed by Van Wyk [37].

A 3D planar polygon is defined by a list of $\eta$ lines (each of which is declared as in Section 6) representing a closed sequence of edges, the plane equation on which it resides, and the color to fill its interior (if it is not hollowed). The formal data type **polygon** is:

```
typedef struct polygon {
        line *l;                "Pointer to an edge list"
        int η;                  "Number of edges in this polygon"
        float plane[4];         "Plane equation coefficients"
        color c;                "Color of the inside of the polygon"
        flag filled;            "Flag indicating whether to fill"
} polygon;
```

If the flag *filled* is **on** both the edges and the interior are drawn, while if the flag is **off** only the edges are drawn. The polygon must be closed and planar (or almost planar), and may be non-simple (may self-intersect) or multiple (may have "holes").

The algorithm to scan-convert a polygon, *POLYGON*, assumes that all edges are in one connected list, and all edges have $x$ and $\Delta x$ variables, which are the current value along the scan line and the increment, respectively. The algorithm begins by filling the interior of the polygon (if the flag *filled* is **on**), followed by scan-conversion of the boundary. This is done to improve converted edge fidelity and to allow the edges to have a different color from the interior. The edges are drawn by the *LINE* algorithm presented in Section 6, while the filling is done by algorithm *POLYGON* described next.

Assume that:

$$Ax + By + Cz + D = 0 \tag{3}$$

is the plane equation of the planar polygon, or the best estimate for the plane equation of the almost planar polygon. The algorithm initially determines the coordinate with the largest plane coefficient (in absolute value), i.e., with the largest polygon projection area onto the principal plane perpendicular to the coordinate axis, and therefore the change in this coordinate is the smallest. Without loss of generality assume that this coordinate is $z$.

The main part of the algorithm *POLYGON* is displayed in Figure 2. Like the common 2D scan-line algorithms, *POLYGON* sorts the polygon edge list along the $y$ direction, and maintains an active edge list of all edges that intersect with the

```
Δz_x = -A / C;
Δz_y = -B / C;                                          "Depth increments"
for each line l in edge_list {                          "Takes O(η)"
        if (l.p1_y > l.p2_y ) exchange l.p1 with l.p2;
        l.Δx = (l.p2_x - l.p1_x ) / (l.p2_y - l.p1_y );
        l.x = l.p1_x ;
        l.Δxz = l.Δx * Δz_x ;
}
Sort edge_list with ascending p1_y ;                    "Takes O(ηlogη)"
Set current line pointer to first in active_edge_list;
z0 = current.x*Δz_x + y_0*Δz_y - D/C + 0.5;             "Initial depth"

for (y = y_0; y ≤ y_f ; y++) {                          "Loop ν times"
        NEW_POS (Y, y);
        for each line l in active_edge_list             "Takes O(θ)"
                l.x += l.Δx ;                           "New edge coordinates"
        Resort active_edge_list using bubble sort;      "Takes O(θ)"

        "Add new edges into active_edge_list:"
        for each line l in edge_list having l.p1_y == y "Takes O(θλ)"
                Enter l into active_edge_list by insert sort;

        "Drop old edges from active_edge_list:"
        for each line l in active_edge_list having l.p2_y ≥ y   "Takes O(θ)"
                Delete l from active_edge_list;

        Set current and last pointers to first and last in active_edge_list;
        z0 += current.Δxz + Δz_y ;                      "Eq. 7"
        z = z0;
        fill = off;
        for (x = FLOOR(current.x); x ≤ last.x; x++) {   "Loop μ times"
                z += Δz_x ;
                if (current.x < x) Toggle fill flag, and advance current line pointer;
                if (fill) {                             "Inside"
                        NEW_POS (X, x);   NEW_POS (Z, FLOOR(z));
                        PUT_VOXEL ();
                }
        }
}
```

Figure 2: *POLYGON* - An Algorithm for Scan-Converting a 3D Polygon

current scan line $y$. For every new $y$, new edges starting at $y$ are merged into the active edge list, and edges in the active list terminating at $y$ are removed. Relying on edge coherence, the $x$ coordinate of each edge on the active list is incrementally updated by the edge's $\Delta x$.

Unlike the 2D case, where runs of pixels are filled between alternating pairs of $x$ coordinates from the active list, in the 3D case the voxels comprising a run are not necessarily having the same depth. Therefore, each voxel along the $x$ direction between alternate members of the active list has to be examined separately, its $z$ coordinate evaluated using the plane equation, and only then written into the CFB.

The algorithm takes unit steps along the $y$ direction between $y_0$ and $y_f$, and for each $y$ it takes unit steps along the $x$, between $x_0$ and $x_f$. $y_0$, $y_f$, and $x_0$, $x_f$ are the extremal values of points along $y$ and $x$, respectively, defining a bounding rectangle. In order to expedite the algorithm, for each $y$ the stepping along $x$ can start on the first (the smallest) $x$ value encountered on the current (sorted) active edge list, and terminate on the last (the largest) $x$ value on that list, rather than stepping along all the $x$ values between $x_0$ and $x_f$. For each $(x,y)$, the algorithm computes the current depth $z$ using the plane equation and then rounds it to an integer coordinate.

The $z$ value for $(x,y)$ is computed by:

$$z = -\frac{A}{C}x - \frac{B}{C}y - \frac{D}{C} \tag{4}$$

Since the smallest change occurs along the $z$ coordinate, $C$ is the largest equation coefficient and $C \neq 0$. As the coefficients of Equation 4 are constants, for each step of the algorithm the evaluation of $z$ requires only two multiplications and two additions. A more efficient approach is to evaluate $z$ incrementally in both directions $x$ and $y$, based upon the value of $z$ in the previous step along $x$ or along $y$, respectively. Given the value $z$ at $(x,y)$, the value $z_{x+1}$ at $(x+1,y)$ is:

$$z_{x+1} = z - \frac{A}{C} = z + \Delta z_x \tag{5}$$

and thus only one addition (subtraction) operation is necessary to evaluate $z$ at $(x+1,y)$ given $z$ at $(x,y)$.

Similarly, given the value $z$ at $(x,y)$ the value $z_{y+1}$ at $(x,y+1)$ can be evaluated with a single addition:

$$z_{y+1} = z - \frac{B}{C} = z + \Delta z_y \tag{6}$$

Since the step in $x$ starts at the $x$ value of the first edge on the active edge list,

Equation 6 cannot be used. Instead, the increment would be along the edge, namely, given the value $z$ at $(x, y)$ by Equation 4, the value $z_0$ at $(x+\Delta x, y+1)$ is:

$$z_0 = -\frac{A}{C}(x+\Delta x) - \frac{B}{C}(y+1) - \frac{D}{C} = z - \frac{A}{C}\Delta x - \frac{B}{C} \tag{7}$$

For each $(x, y)$, the value $z$ is incremented by Equation 5, and if the point is within the polygon (the flag *fill* is **on**), i.e., $x$ is between odd- and even-numbered edges on the active edge list, then the voxel is written to the CFB at coordinates $(x, y, ROUND(z))$. In order to avoid rounding, $z$ is incremented by 0.5 once at initialization and truncation (which actually does not take time) is used thereafter.

Use $\mu$ and $\nu$ to signify the two larger sides, parallel to the major axes, of the box enclosing the polygon. Use $\eta$ for the number of edges in the polygon, $\theta$ for the average number of members on the active edge list, and $\lambda$ for the number of new edges for each $v$. Since $\lambda$ is normally very small [27], and $\theta \leq \eta << \mu$ the algorithm complexity is thus $\mathbf{O}(\eta \log \eta + \nu\mu)$, where the first term is the edge sort and the second term is the count of inner loop execution. In contrast, the 2D polygon scan-conversion algorithm has time complexity of $\mathbf{O}(\eta \log \eta + \nu\theta)$. The term $\nu\theta$ portrays handling runs of pixels which is of course faster than handling single voxels $(\nu\mu)$.

## 8. Scan-Converting 3D Polyhedra

A polyhedron is defined by a list of *η_poly* polygons representing a polygon mesh envelope (surface) which completely encloses some volume. Each polygon is defined as in the previous section. The polyhedron must be closed but not necessarily simple (may self-intersect). The formal data type **polyhedron** is:

```
typedef struct polyhedron {
        polygon *poly_list;      "Polygon list for the polyhedron"
        int η_poly;              "Number of polygons in the polyhedron"
        int η_line;              "Number of lines in the polyhedron"
        int η_point;             "Number of vertices in the polyhedron"
        color c;                 "Color of the polyhedron volume"
        flag solid;              "Flag indicating filled volume"
} polyhedron;
```

If the polyhedron represents just the envelope (*solid* is **off**), then only an algorithm to scan-convert the polyhedron surface is executed. This algorithm scan-converts the polygon mesh one polygon at a time, using the *POLYGON* algorithm introduced in the previous section. Each polygon is scan-converted in the direction which best suits its orientation. A better strategy, however, would be to

scan-convert together all the $x$-oriented polygons (those with the smallest change along the $x$ direction), followed by all the $y$-oriented polygons, and finally all the $z$-oriented ones.

If the polyhedron volume is also to be filled (*solid* is **on**) with the color $c$, then the interior is filled first using the *scan-plan* algorithm *POLYHEDRON*, and only then the bounding polygons must be scan-converted, for the following reasons: (1) when a bounding polygon is slanted more than 45 degrees to the scan plane ($x-y$), scan-conversion is imperfect near the surface if only volume filling is done; and (2) the polygons may have a different color from the polyhedron's interior.

The *scan-plane* algorithm *POLYHEDRON*, is conceptually a 3D extension of the scan-line algorithm *POLYGON*. Unlike the filling of a polygon, filling the volume can be done in any direction and in parallel to an axis. Thus, $z$ is arbitrarily selected as the depth direction and ($x-y$) as the scan plane. A list of active polygons is maintained and all polygons in that list should be sorted for increasing $z$ (depth). The sorting allows us to draw a run (vector) of voxels in the $z$ direction between an odd-numbered polygon and its successor in the sorted list. This vector write can be done in parallel (see Section 2).

For each ($x$,$y$) in the bounding box of the polyhedron, the depth of each polygon on the active polygon list is updated incrementally in the same fashion the $z$ coordinate has been updated in the *POLYGON* algorithm. The list of active polygons is then resorted to account for the case of self-intersecting polyhedra.

The list of active polygons is then used to fill in between alternating pairs on the list. In order to determine whether a polygon is actually active for the current ($x$,$y$), the variable *active* associated with the polygon is set to **in**, otherwise it is **out**. Also, in order to update the polygon depth incrementally, the active variable may assume also either the value **new**, in which case the depth is initially computed from the plane equation, or the value **old**, in which case the depth is updated incrementally.

Like the *POLYGON* algorithm, *POLYHEDRON* also maintains an active edge list. New edges are added and non-active ones are deleted from the active edge list as $y$ increases. Every time an edge is encountered along the scan-line $y$, the value of the *active* variable for each of the two parent polygons of the edge is updated. After the polygons on the active polygon list are properly labeled as either **in** or **out**, a depth-fill (vector write) function fills in between alternating pairs of **in** polygons on the list.

The complexity of the *POLYHEDRON* algorithm is linear in the number of pixels in the polyhedron projection, as the algorithm writes runs of voxels as one entity into the CFB.

## 9. Scan-Converting 3D Curves

A 3D parametric polynomial curve is defined as polynomials of some parameter $t$, one for each $x$, $y$, and $z$. Varying the parameter from 0 to 1 defines all the points on the curve segment. Commonly, polynomials of the 3rd degree (i.e., cubic) are employed. Notations used in this article are:

$$T = [t^3 \;\; t^2 \;\; t \;\; 1] \qquad P = \begin{bmatrix} P_1 \\ P_2 \\ P_3 \\ P_4 \end{bmatrix} \qquad 0 \leq t \leq 1 \tag{8}$$

thus the cubic polynomial describing the curve is the vector $r$:

$$r(t) = T \; P \tag{9}$$

Other representations, which give indications of the geometrical shape, are possible by transforming the primitive basis $T$ into another basis, multiplying by the geometric basis matrix $M$ (a $4 \times 4$ matrix):

$$r(t) = T \; M \; G \qquad G = \begin{bmatrix} G_1 \\ G_2 \\ G_3 \\ G_4 \end{bmatrix} \tag{10}$$

where $G$ is the new geometric vector, and

$$P = M \; G. \tag{11}$$

In this article, curves are assumed to be in Bezier form [2, 3]. There is no loss of generality in this assumption, since all cubic curves have their Bezier representation. In this representation $G_1$ and $G_4$ are the curve endpoints, while $3(G_2 - G_1)$ and $3(G_4 - G_3)$ are the end tangents. The $G_i$ are referred to as the geometric *control points*. The matrix $M$ for the Bezier representation is:

$$M_b = \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \tag{12}$$

The Bezier form has the *convex hull* property: the curve resides wholly within the convex hull (polygon boundary) of the control points $G_i$. Bezier's form has in addition the property of decreasing convex hull volume with curve subdivision. The Bezier form, which is widely used in graphics and geometric design, has also the intuitive feature that the four control points explicitly control the shape of the curve which passes through the first and the fourth control points.

One method for scan-conversion is repeated bisection of the curve until the segments reside within a single discrete point [8]. Another method is using the

curve's 3rd order finite forward difference matrix for a given step size $\epsilon$ along the parameter $t$ [8, 12]. Define:

$$E_\epsilon = \begin{bmatrix} 0 & 0 & 0 & 1 \\ \epsilon^3 & \epsilon^2 & \epsilon & 0 \\ 6\epsilon^3 & 2\epsilon^2 & 0 & 0 \\ 6\epsilon^3 & 0 & 0 & 0 \end{bmatrix} \tag{13}$$

Multiply it by the polynomial coefficients $P$ to get the initial difference vector $\Delta_0$:

$$\Delta_0 = \begin{bmatrix} \Delta^0 f_0 \\ \Delta^1 f_0 \\ \Delta^2 f_0 \\ \Delta^3 f_0 \end{bmatrix} = E_\epsilon \; P \tag{14}$$

Now use it to repeatedly obtain the differences of the next step using the calculations (which are a 3rd order DDA):

$$\begin{aligned} \Delta^0 f_{i+1} &= \Delta^0 f_i + \Delta^1 f_i \\ \Delta^1 f_{i+1} &= \Delta^1 f_i + \Delta^2 f_i \\ \Delta^2 f_{i+1} &= \Delta^2 f_i + \Delta^3 f_i \\ \Delta^3 f_{i+1} &= \Delta^3 f_i \end{aligned} \tag{15}$$

and $\Delta^0 f_{i+1}$ is the next step polynomial value. These calculations require three additions per step and no temporary variables, while the alternative method of utilizing the Horner's rule (from Equation 9) requires three additions and three multiplications per step. Note, that if the same step $\epsilon$ is always used, the $E_\epsilon$ matrix is computed only once.

The formal data type of a 3D **curve** is:

```
typedef struct curve {
        fixed control [3][4];      "Bezier 4 control points for x, y, z (Eq. 10)"
        fixed ε;                   "The required parameter step size"
        color c;                   "Color of curve"
} curve;
```

where the $3 \times 4$ control matrix is the four Bezier control points of a Bezier polynomial curve of the parameter $t$, which varies from 0 to 1. A step size along the parameter $t$ may be given in the variable $\epsilon$. If a step is not specified in $\epsilon$ (e.g., it is zero), the step size should be calculated by the scan-conversion algorithm so as to achieve 26-connectivity.

The algorithm *CURVE*, presented in Figure 3, scan-converts a 3D 3rd degree parametric curve in the parameter $t$ according to the difference equations (Equation 15). The only problem remaining is to guarantee 26-connectivity, that is, guaranteeing that the first difference along any dimension is less than 1 in magnitude. In first approximation, this is the same as:

$$\max_{0 \le t \le 1} \left( \left| \frac{\partial x}{\partial t} \right|, \left| \frac{\partial y}{\partial t} \right|, \left| \frac{\partial z}{\partial t} \right| \right) \le 1 \tag{16}$$

Since the curve parametric representations are 3rd degree polynomials, their derivatives are of the 2nd degree. To find the maximum, the extrema of the derivatives of the parametric equations in the range $0 \le t \le 1$ are found by differentiation. The maximum magnitude, $m$, of these extrema is the inverse of the required algorithm step $\epsilon$. A small term is first added to $m$ (e.g., rounding up), to compensate for the approximation.

Algorithm time complexity is linear with the maximum magnitude $m$, which should come near to the number of painted voxels. This is so for all curves which are not too oscillative, which is reasonable to assume about the Bezier class. Each step of the algorithm loop, i.e., each voxel written into the CFB, takes 10 additions, one floating point test, 3 rounding* of floating point numbers to integer coordinates, and one *WRITE_VOXEL*.

## 10. Scan-Converting 3D Surfaces

A 3D bicubic parametric polynomial surface is defined as cubic polynomials of two parameters $u$ and $v$. As both parameters vary across the (0, 1) range, all the points on the surface patch are defined. The notation used is:

$$U = [u^3, u^2, u, 1] \tag{17}$$

$$V = [v^3, v^2, v, 1] \tag{18}$$

with the algebraic representation:

$$r(u,v) = U \; P \; V^{\dagger} \tag{19}$$

Here we have the same representations that are available for curves:

$$r(u,v) = U \; M \; G \; M^{\dagger} \; V^{\dagger} \tag{20}$$

where $M$ is the same transformation matrix as for curves. Again, we use the Bezier form where $G$ is the $4 \times 4$ control point matrix, and $G_{11}$, $G_{14}$, $G_{41}$, $G_{44}$ are the surface patch "corners". The properties of the Bezier's curves also hold for the Bezier's surfaces. Therefore, the subdivision method can be used for

* Rounding can be replaced by truncation and shifting of the coordinates by 0.5 each at initialization.

† means transposed

Find coefficients *coef* for *x(t), y(t), z(t)* by multiplying the Bezier matrix $M_b$ by the control points *control* ($P = M_b\ G$ , Eq. 11);

**if** *($\epsilon$ == 0)*
{
    Find maximum absolute value of the partial derivatives
    $\partial x/\partial t$ , $\partial y/\partial t$ , $\partial z/\partial t$ in the domain $0 \leq t \leq 1$,
    then set $m$ to the maximum of these maxima;
    $\epsilon$ = *MIN (1, 1/CEILING (m));*
}

Calculate initial difference vectors $\Delta^0 x$ , $\Delta^0 y$ , $\Delta^0 z$ by multiplying the matrix $E_\epsilon$ by the coefficients *coef* (Eq. 14);

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

**for** *($t = 0;\ t \leq 1;\ t \mathrel{+}= \epsilon$)*
{
    *"Now start following curve"*
    $\Delta^0 x \mathrel{+}= \Delta^1 x$ ;  *"Apply Eq. 15 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. 15 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. 15 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 3: *CURVE* - An Algorithm for Scan-Converting a 3D Curve

surfaces too. As for the forward difference method, the difference matrix is obtained by multiplying the difference operator by the $P$ matrix (algebraic representation) [12]:

$$\Delta f_{uv} = E_\delta P E_\epsilon^\dagger \tag{21}$$

where $\delta$ and $\epsilon$ are the step size in the $u$ and the $v$ parameters, respectively.

The formal data type **surface** represents a bicubic surface patch defined by cubic equations of two parameters $u$ and $v$, both vary from 0 to 1:

```
typedef struct surface {
        fixed control[3][4][4];        "Bezier 16 control points for x, y, z"
        fixed δ, ϵ;                    "Required step size along u, v"
        color c;                       "Surface color"
} surface;
```

The $3\times4\times4$ control matrix is the 16 Bezier geometric control points of a bicubic Bezier patch. Parameter step sizes along the parameters $u$ and $v$ may be given in $\delta$ and $\epsilon$, respectively. If the steps are not specified the step sizes should be calculated by the scan-conversion algorithm so as to disallow 6-connected tunnels in the surface.

A bicubic polynomial surface patch in $u$, $v$ can be drawn as a sequence of cubic curves where $v$ is constant, and $u$ is varied from 0 to 1. When scan-converting such a surface we must guarantee lack of 6-connected tunnels. This is achieved if the first difference (in first approximation - the partial differential) of the surface in each of the parameters is bounded by 1 in magnitude (otherwise we renormalize it). Thus, we must find the extrema of the derivatives over the surface patch, $n$ for $u$ and $m$ for $v$, 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 eight-degree equations which is very time consuming, and therefore not acceptable. Bounding the derivatives by using the convex hull property of surfaces in Bezier representation is much simpler. Finding the bound on $\partial r / \partial u$ is as follows. The algebraic representation of a 3rd degree polynomial is given in Equation 19. The $D_u$-differentiating operator for the derivative of the surface with respect to the parameter $u$ is:

$$D_u = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 3 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \tag{22}$$

From Equation 20:

$$\frac{\partial r\,(u\,,v\,)}{\partial u} = U\ D_u\ M_b\ G\ M_b{}^{\dagger}\ V^{\dagger} \tag{23}$$

$$= U\ M_b\ (M_b{}^{-1}\ D_u\ M_b\ G\,)\ M_b{}^{\dagger}\ V^{\dagger}$$

The term in parentheses in Equation 23 is the control point matrix ($G_{\partial r}$) for the $\partial r\,/\partial u$ surface in Bezier representation, thus its $D_u$-differentiating operator in Bezier representation is:

$$D_{u_b} = M_b{}^{-1}\ D_u\ M_b = \begin{bmatrix} -3 & 3 & 0 & 0 \\ -1 & -1 & 2 & 0 \\ 0 & -2 & 1 & 1 \\ 0 & 0 & -3 & 3 \end{bmatrix} \tag{24}$$

Now we find the bound on the derivative as the maximum magnitude of all control points of the $\partial r\,/\partial u$ surface. After finding all the maxima, set:

$$\delta = \frac{1}{n} = \min\left( \frac{1}{\sqrt{3}\ \max(G_{\partial x}\,)}, \frac{1}{\sqrt{3}\ \max(G_{\partial y}\,)}, \frac{1}{\sqrt{3}\ \max(G_{\partial z}\,)} \right) \tag{25}$$

Similarly, calculate $\epsilon$ using the $D_v$ matrices. Note that symmetry provides $D_v = D_u{}^{\dagger}$.

The algorithm *SURFACE* to scan-convert a 3D bicubic parametric surface is presented in Figure 4. It employs the forward differences iteration described before to evaluate the surface as a sequence of cubic curves $r\,(u\,,0)$, $r\,(u\,,\epsilon)$, $r\,(u\,,2\epsilon)$, $\cdots$, each of which is computed using a 1D curve forward difference iteration similar to the one described in the previous section. The computation of the initial values of each of the curves is done by a similar 2D iteration. The algorithm first calculates the algebraic coefficient matrix for each of the three components, by the matrix multiplication of $M_b\ GM_b{}^{\dagger}$, where $G$ is the control point matrix. If $\delta$ and $\epsilon$ are unspecified by the surface data, the algorithm computes them, as well as $E_{\delta}$ and $E_{\epsilon}$.

The difference matrix $\Delta f_{uv}$ is found by Equation 21 for each of its components, i.e., the $4\times4$ matrices $\Delta x_{uv}$, $\Delta y_{uv}$, and $\Delta z_{uv}$. The first column of $\Delta x_{uv}$, for example, contains the initial values $\Delta^0 x_u$, $\Delta^1 x_u$, $\Delta^2 x_u$, $\Delta^3 x_u$ needed to evaluate the $x$ component of the curve $r\,(u\,,v_0)$ for the current $v_0$ ($v_0$ is constant, $u$ varies). The zero column of $\Delta x_{uv}$ is therefore copied into the vector $\Delta x_u$ at the beginning of a loop on $u$. The vector $\Delta x_u$ is used as the $x$ forward differences. Similarly the $\Delta y_u$ and $\Delta z_u$ are utilized for the $y$ and $z$ components. The loop on $u$ evaluates all the voxel positions which are the closest to the curve $r\,(u\,,v_0)$

```
float coef[3][4][4];          "Algebraic coefficients matrix"
float u, v;                   "Surface parameters"
float Δx_u [4];               "Differences along u for x"
float Δy_u [4];               "Differences along u for y"
float Δz_u [4];               "Differences along u for z"
float Δx_uv [4][4];           "2D differences for x"
float Δy_uv [4][4];           "2D differences for y"
float Δz_uv [4][4];           "2D differences for z"

Find algebraic coefficients for x(u,v), y(u,v), z(u,v) using Eq. 19;

if (δ == 0 || ε == 0)
        Find δ, ε as described by Eq. 25;

Find initial difference matrices Δx_uv , Δy_uv , Δz_uv using Eq. 21;

for (v = 0; v < 1; v += ε) {

        Copy column 0 of Δx_uv to Δx_u ;
        Copy column 0 of Δy_uv to Δy_u ;
        Copy column 0 of Δz_uv to Δz_u ;

        for (u = 0; u < 1; u += δ) {

                WRITE_VOXEL(ROUND(Δ^0 x_u ),ROUND(Δ^0 y_u ),ROUND(Δ^0 z_u ),c);
                Δ^0 x_u += Δ^1 x_u ;
                Δ^1 x_u += Δ^2 x_u ;
                Δ^2 x_u += Δ^3 x_u ;
                Same for Δy_u and Δz_u ;
        }

        "Update the difference matrices for the next v:"
        For Δx_uv :   Add column 1 to column 0.
                      Add column 2 to column 1.
                      Add column 3 to column 2.

        Same for Δy_uv and Δz_uv ;
}
```

Figure 4: *SURFACE* - An Algorithm for Scan-Converting a 3D Surface

being approximated. The rounded integer values of $\Delta^0 x_u$, $\Delta^0 y_u$ and $\Delta^0 z_u$ are used as the coordinates of the next voxel.

After scan-converting the curve $r(u, v_0)$ the difference matrices $\Delta x_{uv}$, $\Delta y_{uv}$, and $\Delta z_{uv}$ 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 $r(u, v_0+\epsilon)$.

The algorithm complexity is $\mathbf{O}(n\ m)$ which is linear in the number of painted voxels. Each voxel drawn in the inner loop 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 $v$. Initialization involves the non-integer computation of the coefficients, $\delta$, $\epsilon$, and the initial difference matrices.

## 11. Scan-Converting 3D Circles

Pitteway [28] has pioneered in introducing an efficient midpoint algorithm for displaying conic section curve segments. Each incremental step minimizes the displacement of a point located midway between the next two pixel candidates from the true curve. Horn [16] has employed a similar method to derive the special case of the circle algorithm. Jordan et al. [18] have presented algorithms for conic sections, based upon a two-point method, namely based upon the displacements to the next two pixel candidates. Bresenham's algorithm for circles [6], which has gained wide popularity, has been using a similar two-point technique, but is slightly more efficient than the algorithm by Jordan et al. Bresenham's algorithm is based on incremental calculation of a decision variable for determining the next point closest to the intended circle. It tries to minimize the square of the current positional error. The algorithm uses only integer arithmetic, and requires on the average two tests, four additions, and two shifts per point. This algorithm provides best-fit accuracy for integer radii.

McIlroy [23] has extended previous work to half-integer cases and cases where the square of the radius is an integer. Van Aken and Novak [35, 36] have investigated the midpoint method, which is apparently more accurate in terms of the linear error. Other papers describing scan-conversion algorithms for circles and more general curves have been published in the literature (e.g., [1, 9, 10, 34]).

A 3D circle, optionally filled, is defined by its plane direction cosines, its center point and its radius. The formal data type **circle** is:

```
typedef struct circle {
        fixed p[3];          "x, y & z of center point of circle"
        float k[3];          "Plane direction cosines"
        fixed r;             "Radius"
        color c;             "Circumference & disk color"
        flag filled;         "Disk or just circumference?"
} circle;
```

To simplify the discussion on the circle scan-conversion algorithm we shall assume that the center point is $p$ =(0,0,0). The algorithm *CIRCLE*, presented in Figure 5, which scan-converts the circumference of the circle, however, will shift the converted voxels as to reflect a center other than the origin.

The algorithm starts by finding a major plane which is most nearly parallel to the circle's plane. Assume, for simplicity, that this plane is $x-y$. Let the circle's plane equation be:

$$z = \alpha x + \beta y + \delta \tag{26}$$

The coefficients $\alpha$, $\beta$, and $\delta$ are derived from the three direction cosines. The (geometric) projection of the circumference of the circle on the $x-y$ plane is found. It is an ellipse with the following equation:

$$A\ x^2 + B\ x\ y + C\ y^2 = D \tag{27}$$

The algorithm finds the extremal points of this ellipse:

a) The point of minimal $x$, $(x_0, y_0)$.

b) The point of maximal $y$, $(x_m, y_m)$.

c) The point of maximal $x$, $(x_f, y_f)$.

The point of maximal $y$, for example, is calculated by differentiating the ellipse equation to find the required values $x_m$ and $y_m$:

$$x_m = \pm \sqrt{\frac{D}{4A^2C/B^2 - A}} \tag{28}$$

$$y_m = +2\frac{A}{B}\sqrt{\frac{D}{4A^2C/B^2 - A}} \tag{29}$$

Similarly, we obtain $(x_0, y_0)$ and $(x_f, y_f)$.

The algorithm traces the ellipse in the $x$-$y$ plane, and for each point on the ellipse the corresponding voxel of the circle is written into the CFB. The tracking is done in two parts: the "first" quarter of the ellipse from $(x_0,y_0)$ to $(x_m,y_m)$, and then the "second" quarter from $(x_m,y_m)$ to $(x_f,y_f)$. The other half of the ellipse from $(x_f,y_f)$ to $(x_0,y_0)$ is converted by symmetry.

```
Compute actual plane equation of circle: α, β, δ of Eq. 26;
Compute ellipse parameters A, B, C, D of Eq. 27;
Compute extremal points of ellipse: (x0,y0), (xm,ym), (xf,yf) Eqs. 28, 29;
Ax = A * x0;   Bx = B * x0;   By = B * y0;   Cy = C * y0;
e = Ax * x0 + Bx * y0 + Cy * y0 - D;
z = α*(x0+px) + β*(y0+py) + δ;
for (x=ROUND(x0), y=ROUND(y0); x ≤ xm ) {          "Trace (x0,y0) to (xm,ym)"
      e^0+ = e + C + Bx + 2*Cy;                    "Error for: Δx =0, Δy =1"
      e^++ = e^0+ + A + B + 2*Ax + By;             "Error for: Δx =1, Δy =1"
      e^+0 = e + A + 2*Ax + By;                    "Error for: Δx =1, Δy =0"
      if (e^0+ is the smallest) {   y++;
            e = e^0+;  z += β;  By += B;  Cy += C;
      }
      else if (e^++ is the smallest) {    x++;  y++;
            e = e^++;  z += α+β;  Ax += A;  Bx += B;  By += B;  Cy += C;
      }
      else { x++;                                  "e^+0 is the smallest"
            e = e^+0;  z += α;  Ax += A;  Bx += B;
      }
      WRITE_VOXEL (x, y, ROUND(z), c);
      WRITE_VOXEL (2*px -x, 2*py -y, ROUND(2*pz -z), c);
}
while ( y > ROUND(yf ) ) {                         "Trace (xm,ym ) to (xf ,yf )"
      e^+0 = e + A + 2*Ax + By;                    "Error for: Δx =1, Δy =0"
      e^+- = e^+0 - B + C - 2*Cy - Bx;             "Error for: Δx =1, Δy =-1"
      e^0- = e + C - 2*Cy - Bx;                    "Error for: Δx =0, Δy =-1"
      if (e^+0 is the smallest) {   x++;
            e = e^+0;  z += α;  Ax += A;  Bx += B;
      }
      else if (e^+- is the smallest) {    x++;  y--;
            e = e^+-;  z += α-β;  Ax += A;  Bx += B;  By -= B;  Cy -= C;
      }
      else { y--;                                  "e^0- is the smallest"
            e = e^0-;  z -= β;  By -= B;  Cy -= C;
      }
      WRITE_VOXEL (x, y, ROUND(z), c);
      WRITE_VOXEL (2*px -x, 2*py -y, ROUND(2*pz -z), c);
}
```

Figure 5: *CIRCLE* - An Algorithm for Scan-Converting a 3D Circle

The algorithm starts from point $(x_0, y_0, z_0)$, where $z_0$ is calculated by Equation 26. It substitutes $x_0$, $y_0$ into Equation 27 to obtain $D_0$, and subtracts the parameter $D$ from $D_0$ to obtain the tracking error $e_0$ for step 0:

$$e_0 = A\ x_0^2 + B\ x_0\ y_0 + C\ y_0^2 - D \tag{30}$$

It then moves with steps $\Delta x$, $\Delta y$ where there are three possibilities:

a) $\Delta x = 0, \quad \Delta y = 1$

b) $\Delta x = 1, \quad \Delta y = 0$

c) $\Delta x = 1, \quad \Delta y = 1$

where the error in step $i+1$ is:

$$e_{i+1} = e_i + A\ \Delta x^2 + B\ \Delta x\ \Delta y + C\ \Delta y^2 + \tag{31}$$

$$2Ax_i\ \Delta x + Bx_i\ \Delta y + By_i\ \Delta x + 2Cy_i\ \Delta y$$

The products $By$, $Bx$, $Ax$ and $Cy$ are canceled by keeping their previous value, and adding $A$ if $\Delta x = 1$ and $B$ if $\Delta y = 1$. The other products are by 1 or 0 and are calculated by tests.

The algorithm then selects from the above options the one which minimizes the tracking error in step $i+1$. It updates the tracking error and the products in accord, and also the $z$ value by adding $\alpha$ or $\beta$ to the previous $z$ where appropriate. Finally, it draws the new point $(x, y, z)$ and also $(-x, -y, -z)$ for reasons of symmetry. This tracing of the ellipse continues while $y < y_m$.

The next portion of the ellipse is similarly traced while $x < x_f$, where the step options are:

a) $\Delta x = 0, \quad \Delta y = -1$

b) $\Delta x = 1, \quad \Delta y = 0$

c) $\Delta x = 1, \quad \Delta y = -1$

By choosing a step of magnitude 1 at most in each direction and selecting the projection plane so that the angle between it and the circle's plane is less or equal in magnitude to 45 degrees, we guarantee 26-connectivity. Filling in the circle is performed by drawing 6-connected straight line segments for each $y$ between pairs of points $(x_1, y, z_1)$ and $(x_2, y, z_2)$. The 6-connectivity is required because of rounding errors at the endpoints. A modified circumference drawing algorithm is best used, to facilitate finding the above point pairs, where the points are kept in an array in the order of drawing while the symmetric points are kept in separate arrays. A by-product of the circle algorithm is the ability to use it for a general ellipse, as nowhere did we use the fact that the object is a circle rather than an ellipse.

## 12. Scan-Converting Quadratic Objects

### 12.1. Cylinders

The formal data type **cylinder** is:

```
typedef struct cylinder {
        float axis[3];          "Cylinder axis equation coefficients"
        fixed radius;           "Cylinder radius"
        float plane[4];         "Coefficients of plane of 'bottom' base"
        fixed length;           "Cylinder length. Can be negative"
        color c;                "Color of surface & interior"
        flag filled;            "Fill, or just draw surface?"
} cylinder;
```

The surface of the cylinder is scan-converted as follows. The algorithm first finds the base ellipse equations and their projection on an appropriate major plane, as in Section 11. The projections are ellipses, so the algorithm in Section 11 may be used for stepping along their circumference. The stepping is done along both ellipses synchronously (starting at minimal $x$ on both), and draws a straight line segment between each two new points generated. Finally, it draws the base ellipses using the circle-fill algorithm.

If the cylinder has to be filled, the algorithm, while stepping on both ellipses, fills both base ellipses synchronously and draws 6-connected straight line segments between the relevant point pairs. In order to prevent a phase error between the point pairs, the ellipse center coordinates are rounded.

### 12.2. Cones

The formal data type **cone** is:

```
typedef struct cone {
        float axis[3];          "Cone axis equation coefficients"
        float plane[3];         "Direction cosines of 'bottom' circle"
        fixed length;           "Cone length (can be negative)"
        fixed focal[3];         "x, y & z of cone focal point"
        fixed angle;            "Opening angle of cone"
        color c;                "Color of surface & interior"
        flag filled;            "Fill, or just draw surface?"
} cone;
```

The algorithm to scan-convert the cone, steps along its elliptic base in a way similar to the cylinder algorithm, and draws straight line segments to the cone's focal point. In order to fill in the cone the algorithm fills its base ellipse while

drawing straight line segments from each point to the focal point.

### 12.3. Spheres

The formal data type **sphere** is:

```
typedef struct sphere {
        fixed center[3];        "x, y & z of center of sphere"
        fixed radius;           "Radius of sphere"
        color c;                "Color of surface & interior"
        flag filled;            "Fill this sphere?"
} sphere;
```

Scan-converting a sphere is simpler than either a cone or a cylinder because of the sphere's symmetry. This symmetry allows scan-converting the surface by drawing circles parallel to a major plane (say $x$–$y$), for which Bresenham's circle algorithm [6] can be used.

The sphere is split into two region types: the equatorial region and the pole regions. The equator is the region where the $|z| \leq R/\sqrt{2}$, the poles being the rest of the sphere. Assuming a sphere with a center at the origin and of radius $R$, we draw it in slices parallel to $x$–$y$ (and then re-draw it with slices parallel to the other major planes, to guarantee lack of tunnels). The algorithm only calculates the points with positive $z$, drawing those with negative $z$ by symmetry. It also uses circle symmetry to calculate only one eighth circle in each circle.

The algorithm first calculates for each positive integer $z$ in the equator region the relevant circle radius:

$$r = \sqrt{R^2 - z^2} \tag{32}$$

and draws a circle of radius $r$ with center (0, 0, $z$) using Bresenham's circle algorithm. For each point ($x$, $y$, $z$) drawn, it draws also the point ($x$, $y$, $-z$). If the sphere is to be filled, it draws a voxel-run between all pairs of points ($-x$, $y$, $z$) and ($x$, $y$, $z$). For each positive integer $x \geq R/\sqrt{2}$, and $y = 0$, it finds $z$ by the sphere equation and draws a circle of radius $x$ with center (0, 0, $z$) which draws the pole regions. It draws also the symmetric points (on the other side of the $z = 0$ plane), and fills if required.

The algorithm complexity is linear in the number of voxels painted, where for each circle one square root calculation is necessary (even though this may be omitted by using Bresenham's circle algorithm to calculate $r$ as well), but only additions and left shifts are used inside the inner loop. However, temporal complexity of the sphere filling algorithm, using a voxel-run write mechanism, is the same as for drawing an unfilled sphere.

## 13. Concluding Remarks

We have described 3D scan-conversion algorithms for lines, polygons, polyhedra, curves, surfaces, circles, and quadratic objects, from their geometric $R^3$ representation to $Z^3$ voxel-image space. The conversion was achieved while obeying the fidelity, continuity and efficiency requirements. The algorithms are incremental and use only simple operations within their inner loops.

The polyhedron-fill and sphere-fill algorithms scan-convert with algorithm complexity which is linear in the number of voxels written to the CFB. Their temporal complexity is, however, linear in the number of pixels in the object projection rather than the number of voxels in the object itself. All the other algorithms do scan-conversion with algorithm and temporal complexities which are linear in the number of voxels in the object.

All the 3D scan-conversion algorithms were implemented as part of the GP3 of the CUBE architecture. The GP3 was simulated in software, written in C under UNIX and has been running on VAX computers and SUN workstations. Some of the algorithms, however, may use special purpose hardware (DDAs, 3rd order DDAs, etc.) which has been designed to improve conversion speed. Auxiliary hardware for memory access has been designed and implemented as a custom-built wirewrap board for both single and vector access to the CFB, permitting real-time conversion speeds.

***Acknowledgment***

This work was supported by the National Science Foundation under grant DCR-86-03603. The initial steps of the this research were supported by the National Council for Research and Development, Israel, under grant 283-63.601.

## 14. References

1. Badler, N. I., "Disk Generators for a Raster Display Device", *Computer Graphics and Image Processing*, **6**, 4 (August 1977), 589-593.
2. 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.
3. Bezier, P., *Numerical Control - Mathematics and Applications*, A. R. Forrest (Trans.), Wiley, London, 1972.
4. Bresenham, J. E., "Algorithm for Computer Control of a Digital Plotter", *IBM Systems Journal*, **4**, 1 (1965), 25-30.
5. Bresenham, J. E., "Incremental Line Compaction", *Computer Journal*, **25**, 1 (February 1982), 116-120.

6. Bresenham, J. E., "A Linear Algorithm for Incremental Digital Display of Circular Arcs", *Commun. ACM*, **20**, 2 (February 1977), 100-106.

7. Cederberg, R. L. T., "A New Method for Vector Generation", *Computer Graphics and Image Processing*, **9**, 2 (February 1979), 183-195.

8. 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.

9. Danielsson, P. E., "Incremental Curve Generation", *IEEE Trans. on Computers*, **C-19**, (1970), 783-793.

10. Doros, M., "Algorithms for Generation of Discrete Circles, Rings, and Disks", *Computer Graphics and Image Processing*, **10**, 4 (August 1979), 366-371.

11. Earnshaw, R. A., "Line Tracking for Incremental Plotters", *Computer Journal*, **23**, 1 (February 1980), 46-52.

12. Foley, J. D. and van Dam, A., *Fundamentals of Interactive Computer Graphics*, Addison-Wesley, Reading, MA, 1982.

13. Goldwasser, S. M., "A Generalized Object Display Processor Architecture", *IEEE Computer Graphics & Applications*, **4**, 10 (October 1984), 43-55.

14. Guibas, L. J. and Stolfi, J., "A Language for Bitmap Manipulation", *ACM Trans. on Graphics*, **1**, 3 (July 1982), 191-214.

15. Hersch, R. D., "Descriptive Contour Fill of Partly Degenerated Shapes", *IEEE Computer Graphics & Applications*, **6**, 7 (July 1986), 61-70.

16. Horn, B. K. P., "Circle Generators for Display Devices", *Computer Graphics and Image Processing*, **5**, (June 1976), 280-288.

17. Jackel, D., "The Graphics PARCUM System: A 3D Memory Based Computer Architecture for Processing and Display of Solid Models", *Computer Graphics Forum*, **4**, (1985), 21-32.

18. Jordan, B. W., Lennon, W. J. and Holm, B. D., "An Improved Algorithm for the Generation of Nonparametric Curves", *IEEE Trans. on Computers*, **C-22**, 12 (December 1973), 1052-1060.

19. Kaufman, A. and Bakalash, R., "Memory and Processing Architecture for 3-D Voxel-Based Imagery", submitted for publication, 1986.

20. Kaufman, A. and Bakalash, R., "A 3-D Cellular Frame Buffer", *Proc. EUROGRAPHICS'85*, Nice, France, September 1985, 215-220.

21. Kaufman, A., "Voxel-Based Architectures for Three-Dimensional Graphics", *Proc. IFIP'86*, Dublin, Ireland, September 1986, 361-366.

22. Kaufman, A., "Memory Organization for a Cubic Frame Buffer", *Proc. EUROGRAPHICS'86*, Lisbon, Portugal, August 1986, 93-100.

23. McIlroy, M. D., "Best Approximate Circles on Integer Grids", *ACM Trans. on Graphics*, **2**, 4 (October 1983), 237-263.

24. Newman, W. M. and Sproull, R. F., *Principles of Interactive Computer Graphics*, (2nd ed.), McGraw Hill, New York, 1979.

25. 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.

26. Pavlidis, T., *Algorithms for Graphics and Image Processing*, Computer Science Press, Rockville, MD, 1982.

27. Pavlidis, T., "Scan Conversion of Regions Bounded by Parabolic Splines", *IEEE Computer Graphics & Applications*, **5**, 6 (June 1985), 47-53.

28. Pitteway, M. L. V., "Algorithm for Drawing Ellipses or Hyperbolae with a Digital Plotter", *Computer Journal*, **10**, 3 (November 1967), 282-289.

29. Pitteway, M. L. V. and Green, A. J. R., "Bresenham's Algorithm with Run Line Coding Shortcut", *Computer Journal*, **25**, 1 (February 1982), 114-115.

30. Roberge, J., "A Data Reduction Algorithm for Planar Curves", *Computer Vision, Graphics, and Image processing*, **29**, (1985), 168-195.

31. Rosenfeld, A., "Three-Dimensional Digital Topology", Computer Science Center, Univ. of Maryland, Tech. Rep.-936, 1980.

32. Sproull, R. F., "Using Program Transformations to Derive Line-Drawing Algorithms", *ACM Trans. on Graphics*, **1**, 4 (1982), 259-273.

33. Srihari, S. N., "Representation of Three-Dimensional Digital Images", *ACM Computing Surveys*, **13**, 4 (December 1981), 399-424.

34. Suenaga, Y., Kamae, T. and Kobayashi, T., "A High-Speed Algorithm for the Generation of Straight Lines and Circular Arcs", *IEEE Trans. on Computers*, **TC-28**, 10 (October 1979), 728-736.

35. Van Aken, J. R. and Novak, M., "Curve-Drawing Algorithms for Raster Displays", *ACM Trans. on Graphics*, **4**, 2 (April 1985), 147-169.

36. Van Aken, J. R., "An Efficient Ellipse-Drawing Algorithm", *IEEE Computer Graphics & Applications*, **4**, 9 (September 1984), 24-35.

37. Van Wyk, C. J., "Clipping to the Boundary of a Circular-Arc Polygon", *Computer Vision, Graphics, and Image Processing*, **25**, (1984), 383-392.