Some Practical Aspects of Rendering

Andreas Schilling

ABSTRACT The scan conversion of simple primitives, e.g., vectors and triangles has been worked on in many different ways. General descriptions of algorithms often do not consider 'minor' problems, that can be difficult to solve in practical implementation. Some of these problems are addressed in the following and ways to solve them are presented[1].

The paper consists of two parts. The first part deals with the scan conversion of triangles, the second part describes the implementation of two vector drawing algorithms.

- Drawing Triangles: Calculation of Parameters for Incremental Algorithms

    1. Polygons can be represented by edge functions, that are negative on one, and positive on the other side of the edge. This representation is used in rendering hardware like PROOF [5], Pixel Planes [3] or in software algorithms like the one described by Pineda [4]. The parameters can be chosen in such a way, that the function represents the distance between pixel and edge. In this case the function value can be used to determine the subpixel mask. To get these parameters, the function has to be normalized [1], which usually requires the calculation of a square root. But there is an elegant way to avoid the square root.
    2. Color increments (Gouraud shading) and z increments, that are used to interpolate the colors and depth can become very large, if they are calculated in the conventional way, which is described e.g. in [2]. This can cause the color (z) value of edge pixels to be computed wrong or even to overflow. It is shown, how this occurs and how the problem can be solved.

- Drawing Lines

    Line drawing without antialiasing is performed with the Bresenham algorithm, which can be implemented on the triangle render hardware.

    A simple antialiasing is performed with an algorithm, that is not much more complex than the Bresenham algorithm.

[1]The experiences, described here were gained in a research project, partly supported by the Commission of the European Communities through the ESPRIT II-Project SPIRIT, Project No. 2484 [6,7].

delivered by
EUROGRAPHICS DIGITAL LIBRARY
www.eg.org diglib.eg.org

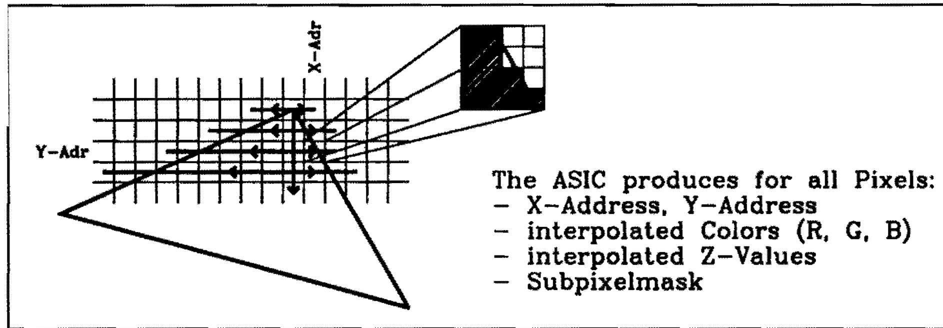

Fig. 1. Functions of the ASIC

1 Drawing Triangles: Calculation of Parameters for Incremental Algorithms

The rendering chip in the SPIRIT workstation is able to render triangles and lines. The first information that is needed for the rendering of triangles (see Fig. 1) is the information, which pixels belong to the triangle. How to get these pixels is the topic of the first section.

1.1 A Practical Distance Measure — the Square Distance (L_1 Norm)

For the scan conversion, we use an algorithm like the one described by Pineda in [4]. A presumption for this kind of algorithms is an edge function, that behaves like the one shown in Fig. 2. It is positive on one side and negative on the other side of the edge. With three units that can calculate such edge functions, we can now decide, whether a certain point lies inside the triangle or outside. If all three edge functions are positive, the point is inside, otherwise it is outside.

How Can We Get Such a Function?

The easiest way is to choose a linear function

$$E(x,y) = (x - X)de_x + (y - Y)de_y$$

with the condition:

$$de_x \Delta X + de_y \Delta Y = 0$$

If we use

$$de_x = \Delta Y$$

as X increment and

$$de_y = -\Delta X$$

as increment in Y direction, we get the formula suggested by Pineda with the advantage that the calculation is very simple. The edge units can be built of only adders, without multipliers, as we only have to add the increments proceeding from one pixel to its neighbor.

We can scale the above formula by an arbitrary factor. So if we need the Euclidean distance, we can normalize the increments by dividing the values by the Euclidean length of the vector (L_2 norm). We then get the following increments:

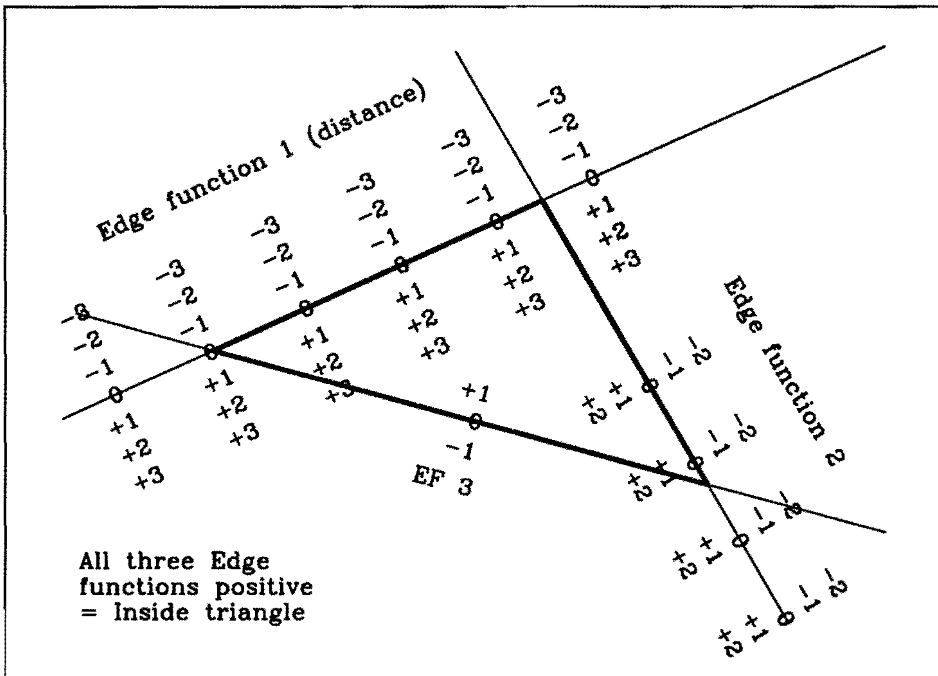

Fig. 2. Example of edge functions for rendering

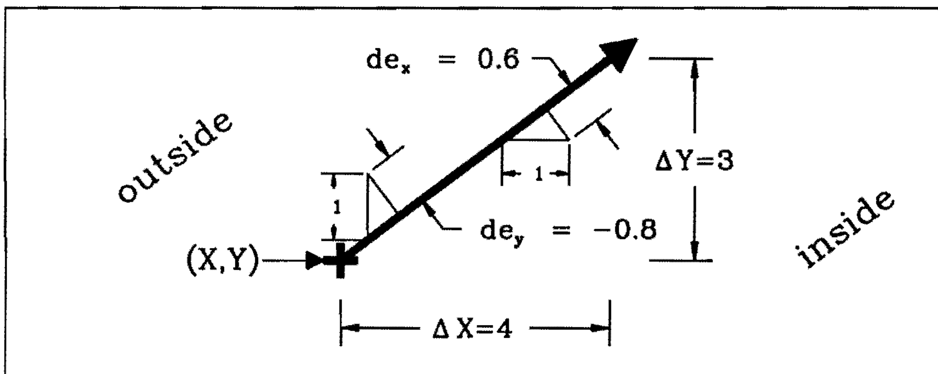

Fig. 3. *X* and *Y* increments of the edge function

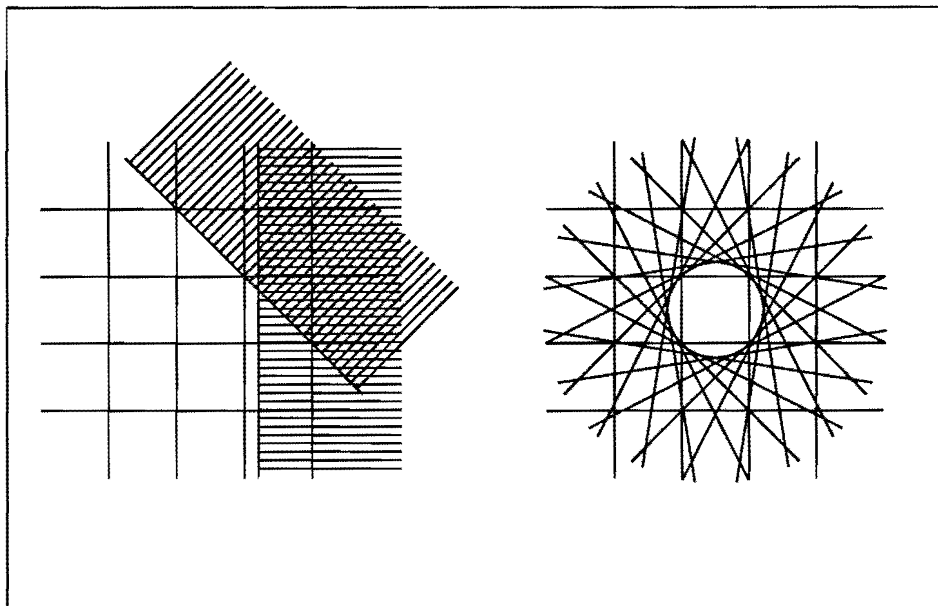

Fig. 4. Circular distance

$$de_x = \frac{\Delta Y}{\sqrt{\Delta X^2 + \Delta Y^2}}$$

and

$$de_y = -\frac{\Delta X}{\sqrt{\Delta X^2 + \Delta Y^2}}$$

and we can still use the same edge units, because the distance is a linear function in X and Y (see example edge in Fig. 3).

Why Do We Need This Distance?

Until now, we used only the sign of the edge function for the decision if we are in or out. So the value of the distance is of no interest. But if we want to calculate subpixel information for later antialiasing, we need exact data about the edge. The distance, together with the slope information is enough to look up the subpixel mask, i.e. the information, which part of the pixel is covered (see Fig. 1). One little detail has to be noticed. We now have to consider not only pixels with their center inside the polygon (positive edge function), but also pixels that are covered less than half (edge function between 0 and –0.something). We cannot give a fixed distance, because it is different for edges with different slopes ($1/\sqrt{2}$ for edges with a slope of 45°, $1/2$ for vertical or horizontal edges). So if we take all pixels not more than $1/\sqrt{2}$ away from the edge into consideration, we will get too many pixels, but that is better than losing pixels that we wanted to get. In Fig. 4 can be seen, why we call this distance the circular distance. All edges that have a given distance from the pixel center form a circle.

Now there is a formula for the increments that solves several problems at one time. If we look at the most demanding part of the increment calculation above, we see the square root in that formula. Now the simplest solution is to omit the root and take the

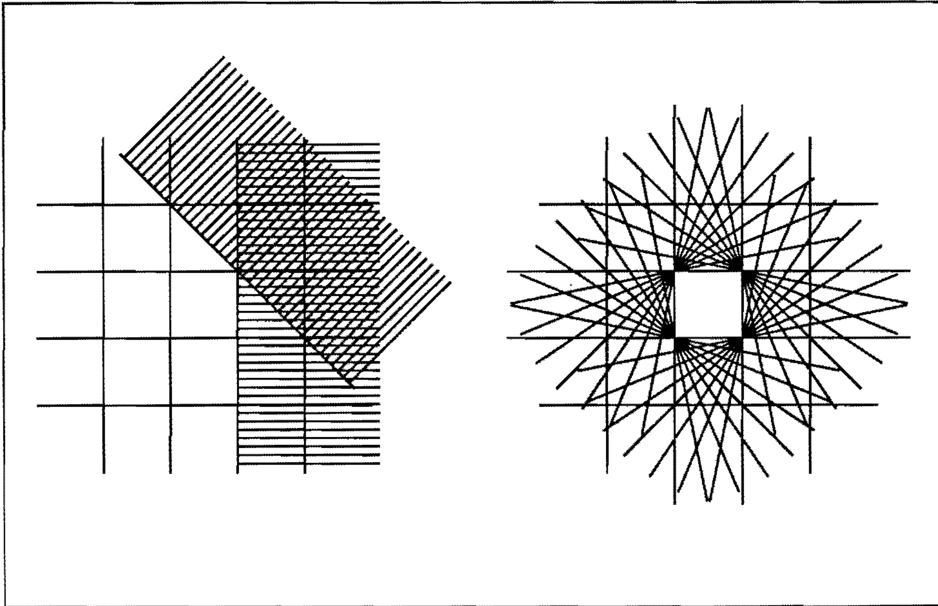

Fig. 5. Square distance

sum of the absolute values of ΔX and ΔY instead. Speaking in mathematical terms that means, we divide by the L_1 norm or Manhattan distance instead of the L_2norm. The new increments are:

$$de_x = \frac{\Delta Y}{|\Delta X| + |\Delta Y|}$$

and

$$de_y = -\frac{\Delta X}{|\Delta X| + |\Delta Y|}$$

The distance is not independent of the angle anymore. But if we don't want to do a more complex filtering (like convolution with sine(dist)), than we need something like a rectangular box filter. A circular filter never would result in a homogeneous coverage of the screen. So this formula is not only more easy to calculate, but also a more desired result. We call this definition of distance the square distance (see Fig. 5).

For the calculation of the subpixel mask, the square distance is as useful as the circular one, because all information about the edge is contained in the distance and the increments (for the slope of the edge).

1.2 Color and z Increments

Why Do We Need Increments?

Let's explain it with the calculation of the depth. We have a plain triangle in the x-y-z space. We could calculate the depth from the x and y coordinates with a linear formula like $z = dz_x \times x + dz_y \times y + z_0$. But normally we render neighboring pixels one after the

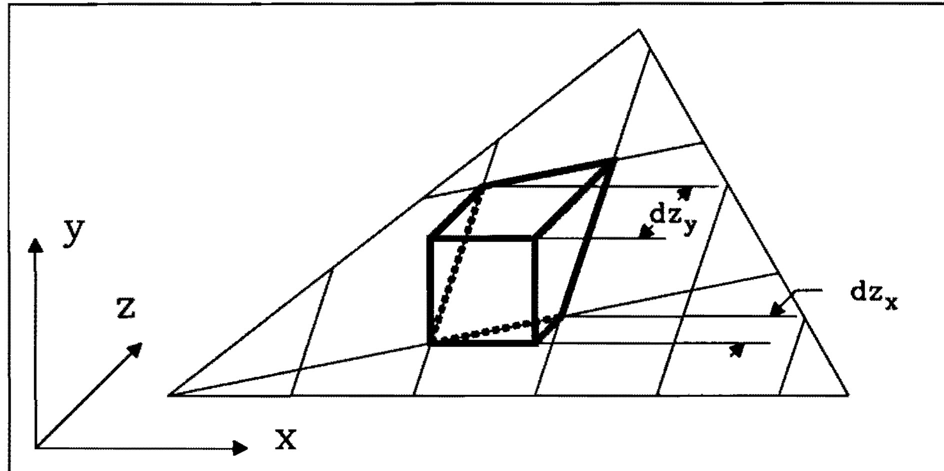

Fig. 6. Calculation of z with an incremental algorithm

other, so we can save the multiplication and add only dz_x or dz_y as we go from one pixel to the next (see Fig. 6).

How Can We Calculate the Increments?

Because we have the three vertices of the triangle with their x, y and z values, it is as simple as solving a system of linear equations.

The formula for dz_x for example is:

$$dz_x = \frac{z_1 y_2 - z_2 y_1}{x_1 y_2 - x_2 y_1} = \begin{vmatrix} z_1 & z_2 \\ y_1 & y_2 \end{vmatrix} / \begin{vmatrix} x_1 & x_2 \\ y_1 & y_2 \end{vmatrix}$$

(If vertex V_0 is not located at the origin, we move it to the origin by subtracting x_0, y_0 and z_0 from the coordinates of the vertices V_1 and V_2)

But let's have a closer look at what we really do (Fig. 7). First the two-fold area of the triangle in the x-y-plain is calculated ($A_{xy} = \begin{vmatrix} x_1 & x_2 \\ y_1 & y_2 \end{vmatrix}$). Then the two-fold area of the triangle in the y-z-plain is divided by the first value. ($A_{zy} = \begin{vmatrix} z_1 & z_2 \\ y_1 & y_2 \end{vmatrix}$). The result is the increment in the x-direction. The values for the y-direction and the color increments for both directions are obtained analogously. Because the area in the x-y-plain may be very small, the increments can become very large.

What Is the Problem with Large Increments?

First of all, big increments don't fit into our registers. Second, and that is the main problem, we can get very wrong z values or colors. Fig. 8 shows, how this happens. The problem is, that with our equations we calculate the z or color at the pixel center. So if we need the z value or color of a pixel, that lies on the border of the triangle with the center outside of it, we get a value that can be even outside of the allowed range.

In order to avoid this, we apply the following rule: The area A_{xy} is divided by the longer side of its bounding box. If the result is smaller than 1, this means that in the average in every line (column) less than half a pixel is covered. Surely not more than one pixel

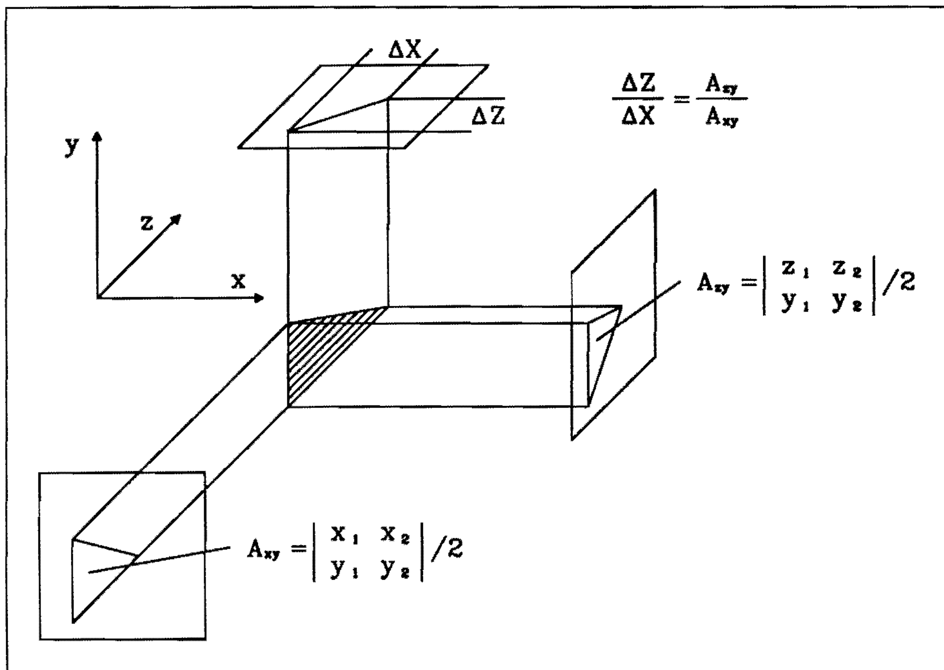

Fig. 7. Calculation of increments

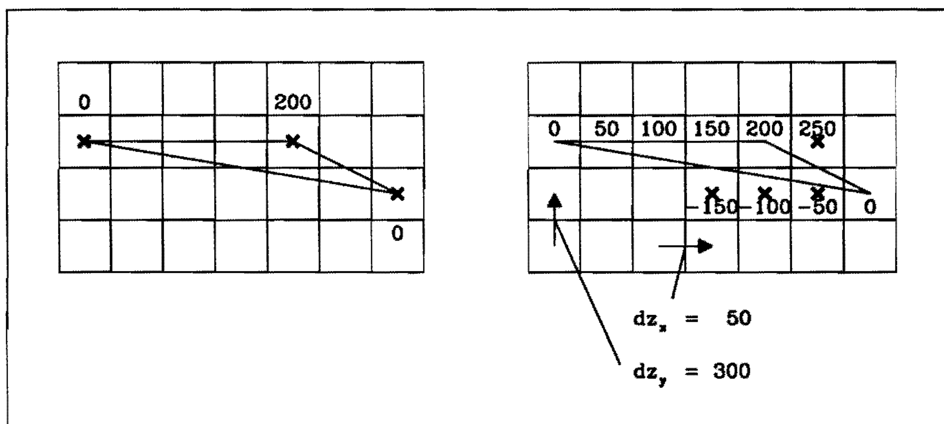

Fig. 8. Overflow problem with conventional increments

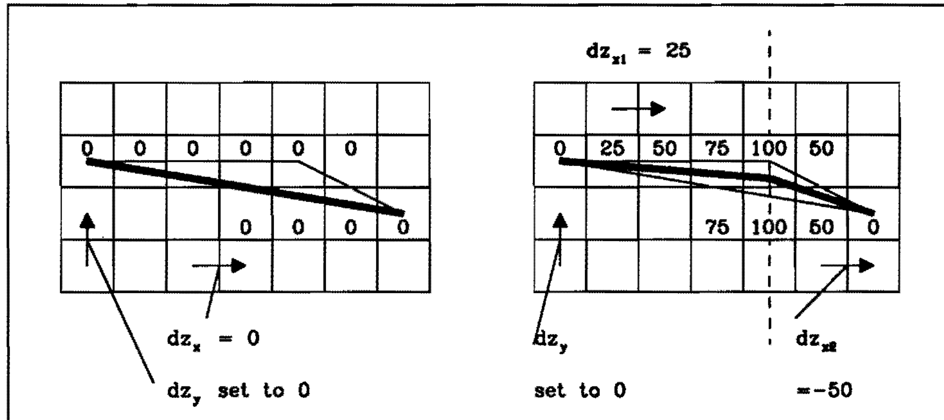

Fig. 9. Solutions for the overflow problem—lines of interpolation are emphasized

per line (column) is covered. So it doesn't make sense to calculate the increments in the direction perpendicular to that longer side. These increments are set to 0. The increments in the other direction are calculated as the ratio of the edge values of z (respectively r, g or b) and the length of the bounding box. By using this method, the increment values can become not larger than the maximum values of z, r, g and b. With this simple method we introduce other errors by omitting the information of the third vertex. If we want to be totally correct, we can obtain it by interpolating along a line within the triangle as shown in Fig. 9. The formula for the increments looks like $dz_x = (z_1/y_1 + z_2/y_2)/2$ for the first part of the triangle and $dz_x = (z_1/y_1 + z_2/y_2)/2$ for the second part. The disadvantage of the second method is, that the triangle usually must be rendered in two parts.

In both cases we need only 2 protection bits for the digits left of the decimal point (one for overflow, one for underflow). Clipping the output at the minimum and maximum values of the colors or z is still required for the triangles rendered in the normal way.

2 Vector Drawing

Two algorithms are used for vector drawing. Both are implemented on the same hardware, that is used to calculate the edge functions for triangles. For fast lines without antialiasing, we use the Bresenham algorithm. The second algorithm performs the drawing of thin, smoothened vectors, which consist of only two pixels per row (resp. column).

2.1 Fast Vectors

The Bresenham algorithm is easy to perform with the same hardware, that is used for the calculation of the edge functions for the triangles. It is explained using a vector with a slope $0 <= m <= 1$ as an example. Using the Bresenham algorithm a modified distance is calculated. It is the Euclidean distance multiplied by the two-fold length of the vector. However, this distance is calculated one pixel in advance in order to be able to decide how to proceed. Moreover, the calculated value is the (scaled) distance from a point located 1/2 pixels higher than the point exactly right of the current pixel. By this the sign of this value can be used to discern whether to proceed to the right or in the upper right direction.

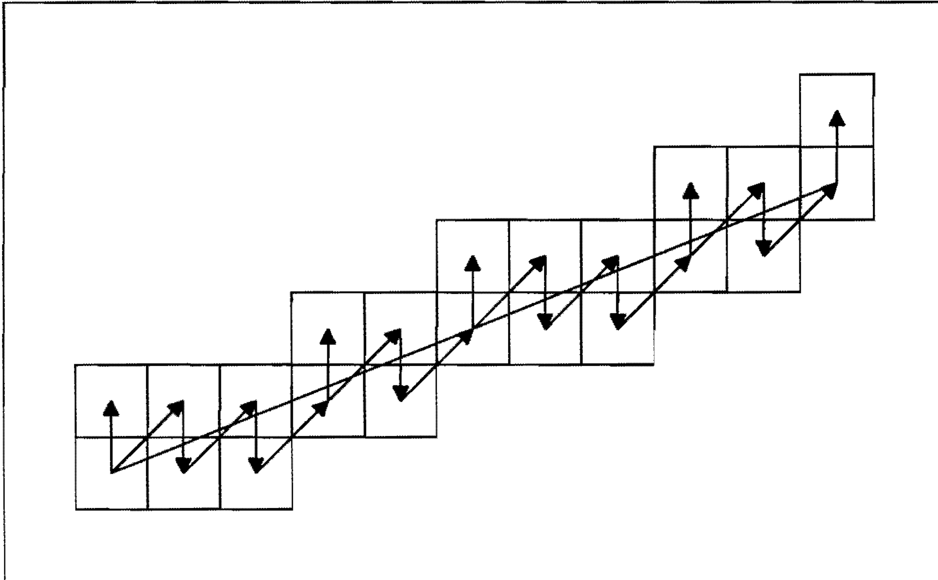

Fig. 10. Thin smoothened vector

So the distance is initialized with a value of $e_0 = \Delta Y - \Delta X/2$, then the increments for the X-direction ($de_x = \Delta Y$) and for the direction 45 degrees up to the right ($de_x + de_y = \Delta Y - \Delta X$) are loaded. The control logic controls — according to the sign of the current distance — whether to increment only X or X *and* Y. Our hardware only allows to load increments up to an absolute value of 1. Therefore the above mentioned increments are divided by an arbitrary power of two, which means, that the number is simply loaded; the hardware does not use the position of the 'decimal'-point. With the Bresenham algorithm only those pixels are calculated that are really needed.

2.2 Thin Vectors with Fast Antialiasing: the Fineline Algorithm

A cheap antialiasing method for vectors uses only two pixels per row resp. column. The brightness values assigned to these two pixels add up to 100% of the desired brightness for the vector. To get such 'antialiased' lines, we can also use the same hardware as for the calculation of the triangle edges. As example we take again a vector with a slope between 0 and 1. Possible directions for the next pixel to calculate are diagonally up to the right, upwards and downwards (see Fig. 10). The values for the distance, the colors and the z-value are stored every time after a step downwards or up to the right. So always the lower of the two pixels of a column is chosen. By this we can ensure, that proceeding to the upper right again leads to a hit.

The brightness of the pixels is distributed in such a way, that the values of one column add up to 100%. For that purpose the up-down-distance from the vector for Pixel i is calculated as:

$$e_i = e_0 + (X_i - X_0) * de_x + (Y_i - Y_0) * de_y,$$

which is achieved with the following parameters:

$$de_x = -\frac{\Delta Y}{\Delta X}$$

and

$$de_y = 1$$

This linear formula implies, that the distance function e for a neighboring pixel can be computed by simply adding or subtracting de_y or $de_y + de_x$ resp. for diagonal neighbors. The starting value of the distance e_0 is 0, when the starting point (x,y) of the vector is in a pixel center. Otherwise, e_0 has to be calculated:

$e_0 = Y - y + (X - x) * de_x$,

where x and y are the coordinates of the vector starting point, and X and Y are the (integer) pixel coordinates of the starting pixel.

The resulting distance is used to calculate the color C of the pixels from the original color C_{org}:

$C = C_{org} * (1 - abs(e))$

When proceeding to a new column, the first pixel is always set. The sign of e determines, whether the second pixel is above or below the first one.

The ASIC works as a state machine. Fig. 11 and 12 show the state diagrams for the Bresenham machine and the Fineline machine. The value e, that is used for the decision, is the distance or error term, described above. The bold arrows indicate the directions, in which the machine proceeds for the case of a line with a slope between 0 and 1. The words PUSH and POP show, where the current value of the error term (or edge function) and colors/z are stored and retrieved in order to proceed from a previously reached point rather than from the current point. Note that the Fineline machine needs only 2 states more than the Bresenham machine. This is not complex compared to the state machine for the Pineda algorithm, which is shown in Fig. 13 for comparison.

Like with the Bresenham algorithm, with the Fineline algorithm normally only those pixels are calculated, that are really needed. This means, that the efficiency remains the same; the Fineline algorithm is exactly two times slower than the Bresenham algorithm, because the number of pixels that have to be drawn has doubled.

3 Conclusion

Some special problems in rendering have been shown and solutions were presented. They represent only a small fraction of the large class of problems, that are generally not addressed in the literature.

References

[1] Fuchs, H., Goldfeather, J., Hultquist, J. P., Spach, S., Austin, J. D., Brooks, F. P., Eyles, J. G. and Poulton, J.: Fast spheres, shadows, textures, transparencies, and image enhancements in pixel-planes. *Computer Graphics*, 19(3):111–120, July 1985.

[2] Fuchs, H. and Poulton, J.: Pixel planes: A vlsi-oriented design for a raster graphics engine. *VLSI Design*, 3:20–28, 3rd Quarter 1981.

[3] Fuchs, H., Poulton, J., Eyles, J., Greer, T., Goldfeather, J., Ellsworth, D., Molnar, S., Turk, G., Tebbs, B. and Israel, L.: Pixel-planes 5: A heterogeneous multiprocessor graphics system using processor-enhanced memories. *Computer Graphics*, 23(3):79–88, July 1989.

[4] Pineda, J.: A parallel algorithm for polygon rasterization. *Computer Graphics*, 22(4):17–20, August 1988.

[5] Schneider, B.: *Eine objektorientierte Architektur für Hochleistungs-Display-Prozessoren*. PhD thesis, Eberhard-Karls-Universität Tübingen, 1990.

[6] Slater, M.: The graphics subsystem of the Spirit workstation. Presentation given at the Eurographics Conference., September 1989.

[7] Zabolitzky, J. G.: The matter of Spirit. Presentation given at the Eurographics Conference., September 1989.

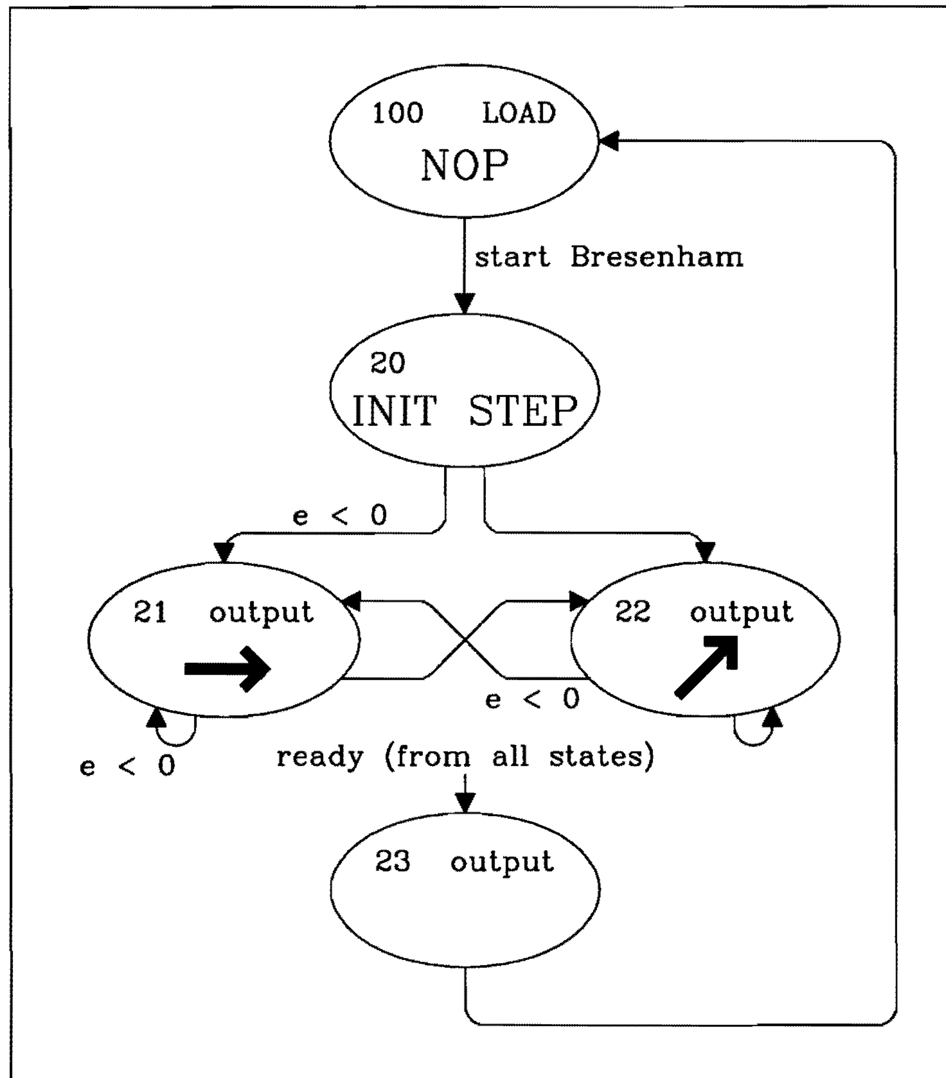

Fig. 11. State diagram for Bresenham vector drawing

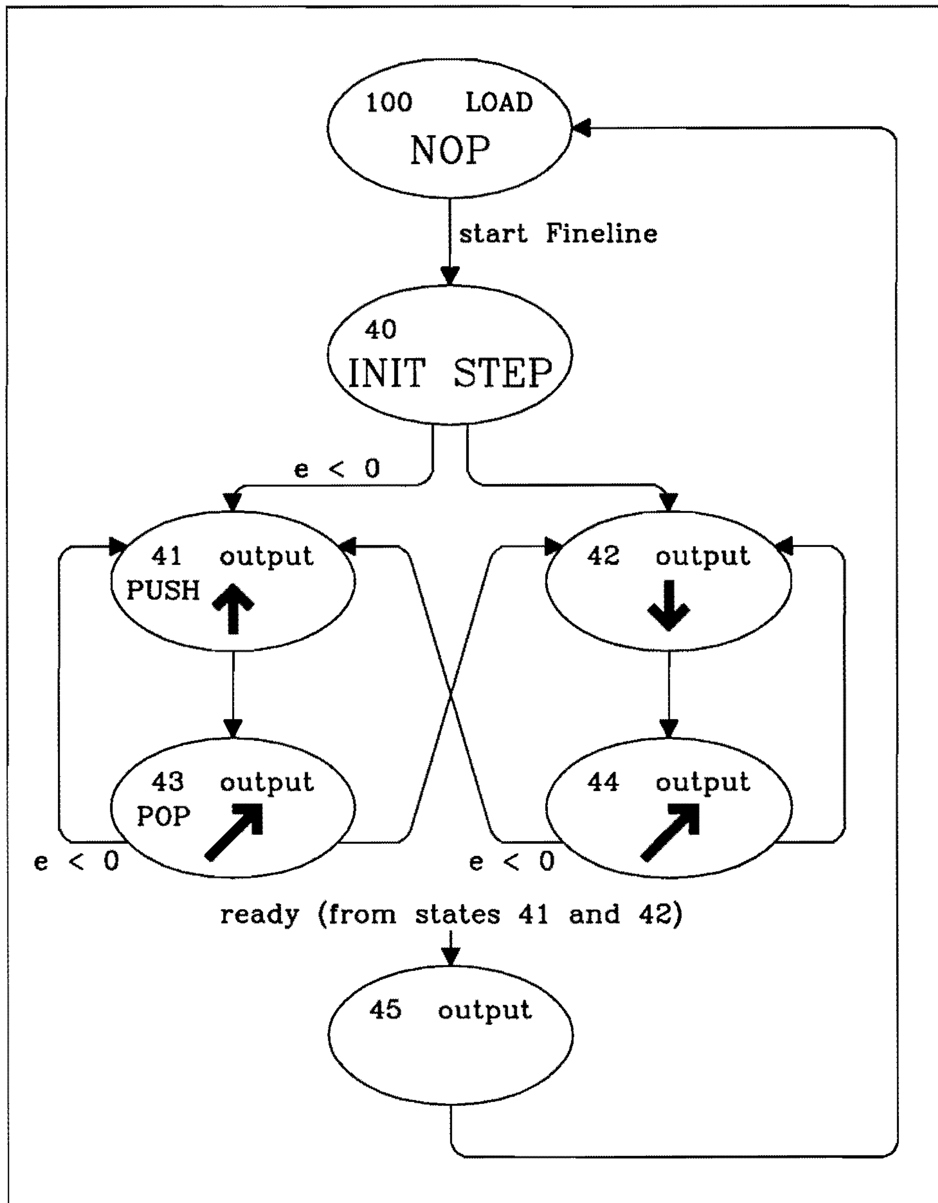

Fig. 12. State diagram for 'antialiased' vector drawing

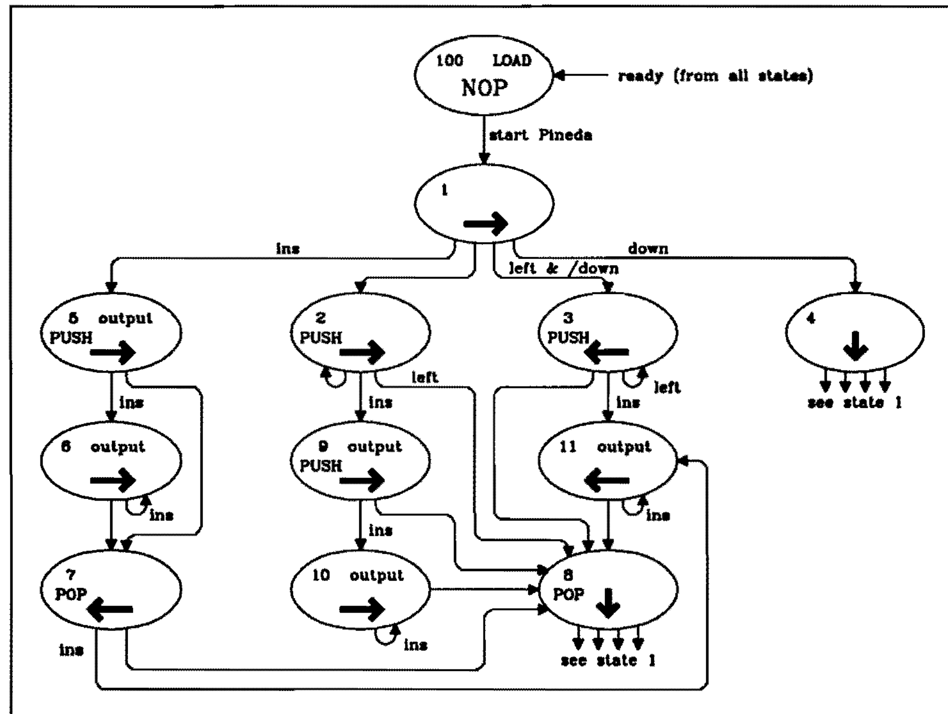

Fig. 13. State diagram for the Pineda algorithm