# On Hardware Implementation of Scan-conversion Algorithms

**László Szirmay-Kalos and Gábor Márton**

Department of Process Control, Technical University of Budapest,

Budapest, Műegyetem rkp. 11, H-1111, HUNGARY

szirmay@fsz.bme.hu

**ABSTRACT**: This paper summarizes a general methodology for the hardware realization of graphics, mainly scan-conversion algorithms. Examples taken to demonstrate the method include anti-aliased and depth-cueing 3D lines and shaded triangles.

**Key Words:** Scan-conversion, incremental concept, hardware implementation

## Introduction

Image synthesis algorithms are expected to be fast and efficient. Since the time allowed to compute the color of a single pixel is often less than a single cycle of a commercial memory, processors, which execute programs by reading the instructions and the data from memories, are far too slow for this task, thus special solutions are needed, including:

1. **Parallelization**, implying the application of many computing units running parallely, and distributing the computational burden among the parallel processors.
2. **Hardware realization**, implying the design of a special digital network instead of the application of general purpose processors, where information about the algorithm is contained by the architecture of the hardware, not by a separate software component as in general purpose systems.

In this paper the general aspects of the hardware realization of graphics, mainly scan conversion algorithms that are responsible for pixel level operations, are discussed. Strictly speaking, hardware realization means a special, usually synchronous, digital network designed to determine the pixel data at the speed of its clock signal. To describe the difficulty of the realization of a function $f$ having integer domain as a combinational network by a given component set, a measure, called **combinational complexity**, is defined as the number of components needed to realize this function. One possible respective component set contains the functional elements of MSI and LSI circuits, including:

- **Adders, combinational arithmetic/logic units** for arithmetical operations,
- **Multiplexers** to realize *then* ... *else* ... branching,
- **Comparators** which generate logic values for the *if* type decisions,
- **Logic gates** which are used for simple logic operations and decisions,
- **Decoders and memories** to realize arbitrary functions over small domains.

## Single-variate functions

Suppose that functions $f_1(k), f_2(k), \ldots, f_n(k)$ having integer domain have to be computed for the integers of an interval between $k_s$ and $k_e$. A computer program carrying out this task might look like this:

> **for** $k = k_s$ **to** $k_e$ **do**
> $\quad F_1 = f_1(k);\ F_2 = f_2(k);\ \ldots F_n = f_n(k);$
> $\quad$ **write**( $k, F_1, F_2, \ldots F_n$ );
> **endfor**

If all the $f_i(k)$-s had low combinational complexity, then an independent combinational network could be devoted to each of them, and a separate hardware counter could generate the consecutive $k$ values, making the hardware realization complete. This usually fails for scan conversion algorithms due to their high combinational complexity. Fortunately, there is a technique, called **incremental concept**, which has proven successful for many scan conversion algorithms. According to the incremental concept, in many cases it is much simpler to calculate the value $f(k)$ from $f(k-1)$ by a function of increment, $\mathcal{F}$, instead of using only $k$. If this $\mathcal{F}$ has low complexity, it can be realized by a reasonably simple combinational network. In order to store the previous value $f(k-1)$, a register has to be allocated, and a counter has to be used to generate the consecutive values of $k$ and to stop the network after the last value has been computed. Generally the calculation of $\mathcal{F}$ requires heuristics. Systematic approaches, however, are available if $f(k)$ can be regarded as the restriction of a differentiable real function $f_r(r)$ to integers, since in this case $f_r(k)$ can be approximated by Taylor's series around $f_r(k-1)$:

$$f_r(k) \approx f_r(k-1) + \frac{df_r}{dk}|_{k-1} \cdot \Delta k = f_r(k-1) + f_r'(k-1) \cdot 1 \tag{1}$$

Unfortunately $f_r'(k-1)$ is usually not an integer, nor is $f_r(k-1)$, and it is not possible to ignore the fractional part, since the incremental formula will accumulate the error to an unacceptable degree. The values of $f_r(k)$ should rather be stored temporarily in a register as a real value, the computations should be carried out on real numbers, and the final $f(k)$ should be derived by finding the nearest integer from $f_r(k)$. Since the realization of floating point arithmetic requires complex digital systems, non-integers must be represented in **fixed point form** where the low $b_F$ bits of the code word represent the fractional part. The number of bits in the fractional part has to be set to avoid incorrect $f(k)$ results due to the cumulative error in $f_r(k)$. If the size of the domain is $N$, then the required length of the fractional part is $\log_2 N$. When the results are converted to integers, instead of finding the nearest integer, truncation ($Trunc$) can be used if 0.5 is added to the number to be converted. The incremental approach is especially efficient if the functions to be calculated are linear, since that makes $f'(k-1) = \delta f$ a constant parameter, resulting in a network of figure 1. The transformed program for linear functions is:

> $F_1 = f_1(k_s) + 0.5;\ F_2 = f_2(k_s) + 0.5;\ \ldots F_n = f_n(k_s) + 0.5;$
> $\delta f_1 = f_1'(k);\ \delta f_2 = f_2'(k);\ \ldots \delta f_n = f_n'(k);$
> **for** $k = k_s$ **to** $k_e$ **do**
> $\quad$ **write**( $k, Trunc(F_1), Trunc(F_2), \ldots, Trunc(F_n)$ );
> $\quad F_1 \mathrel{+}= \delta f_1;\ F_2 \mathrel{+}= \delta f_2;\ \ldots F_n \mathrel{+}= \delta f_n;$
> **endfor**

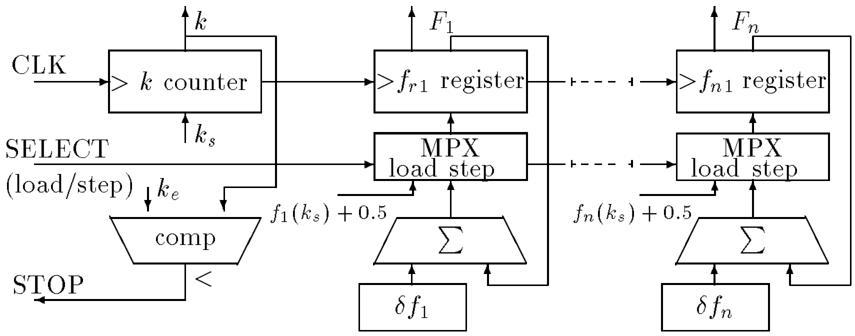


Figure 1: Hardware for linear functions

# Example 1: Drawing simple lines

The simplest application of this method is the DDA line generator (DDA stands for Digital Differential Analyzer). Suppose that the generator has to work for those line segments which satisfy $x_1 \leq x_2$, $y_1 \leq y_2$ and $x_2 - x_1 \geq y_2 - y_1$. Line segments meeting this requirement can be approximated by $n = x_2 - x_1 + 1$ pixels having consecutive $x$ coordinates. The $y$ coordinate of the pixels can be calculated from the equation of the line:

$$y = f(x) = \frac{y_2 - y_1}{x_2 - x_1} \cdot (x - x_1) + x_1 = m \cdot x + b \tag{2}$$

Applying the incremental concept we can get rid of the difficult to implement operations, since $f(x) = f(x) + f'(x-1) = f(x) + m$. The slope of the line $m = (y_2 - y_1)/(x_2 - x_1)$ has to be calculated only once and before inputting it into the hardware.

For those linear functions where the fractional part is not relevant for the next phases of the image generation, the method can be even further optimized by reducing the computational burden of the initialization phase as well. In this case the only purpose of the fractional part is to determine when the integer part has to be incremented (or decremented) due to the overflow caused by the cumulative increments $\delta f$. From this perspective, the fractional part can also be regarded as an error value showing how accurate the integer approximation is. The error value, however, is not necessarily stored as a fractional number. Other representations, not requiring divisions during the initialization, can be found, as suggested by a method of **decision variables**.

Let the fractional part of $f$ be denoted by $fract$ and assume that the increment $\delta f$ is generated as a rational number defined by a division, that is $\delta f = K/D$, whose elimination is the goal of this approach. The overflow of $fract$ happens when $fract + \delta f > 1$. Let the new error variable be $E = 2D \cdot (fract - 1)$, requiring the following incremental formula for each cycle:

$$E(k) = 2D \cdot (fract(k) - 1) = 2D \cdot ([fract(k-1) + \delta f] - 1) = E(k-1) + 2K \tag{3}$$

The recognition of the overflow is based on: $fract(k) \geq 1.0 \quad \Longrightarrow \quad E(k) \geq 0$. If overflow happens, then the fractional part is decreased by one, since the bit having the first positional significance overflowed to the integer part:

$$fract(k) = [fract(k-1) + \delta f] - 1 \quad \Longrightarrow E(k) = E(k-1) + 2(K - D) \tag{4}$$

Finally, the initial value of $E$ comes from the fact that $fract$ has to be initialized to 0.5, resulting in $fract(0) = 0.5 \implies E(0) = -K$. Examining the formulae of $E$, we can conclude that they contain integer additions and comparisons, eliminating all the non-integer operations. A line generator, using this decision variable method, has been proposed by Bresenham [Bre65]. Having made the necessary substitutions, $K = y_2 - y_1$ and $D = x_2 - x_1$, the code of the algorithm in the first octant of the plane is:

**BresenhamLine**$(x_1, y_1, x_2, y_2)$
  $\Delta x = x_2 - x_1$; $\Delta y = y_2 - y_1$;
  $E = -\Delta x$; $dE^+ = 2(\Delta y - \Delta x)$; $dE^- = 2\Delta y$;
  $y = y_1$;
  **for** $x = x_1$ **to** $x_2$ **do**
    **if** $E \leq 0$ **then** $E$ += $dE^-$;
    **else** $E$ += $dE^+$; $y$++;
    **write**$(x, y, color)$;
  **endfor**

# Example 2: Anti-aliased line

From information or signal processing point of view, scan conversion can be regarded as an approximation of the intended image by digital and discrete data. Since the intended world model is continuous, this step always involves an analog-digital conversion. The real world is never bandlimited, because objects appear suddenly (like step functions), introducing infinitely high frequencies. Thus, the sampling theorem can never be satisfied in computer graphics, causing the repeated spectra of the color distribution to overlap, and destroying even the low frequency components of the final spectrum. This phenomenon is called **aliasing**. Aliasing causes the well known phenomenon of stair-like edges or jaggies in raster graphics systems. In order to eliminate, or at least reduce, the effects of aliasing, the model sampled must be band-limited by filtering out the high frequency components. The filter process is usually realized by a convolution with the impulse response of the filter network. Since the ideal low-pass filter's impulse response is the $sync$ function which is difficult to convolve with and can cause negative colors, some approximations are used instead. The simplest approximation is the box filter whose impulse response is:

$$f(x,y) = \begin{cases} 1 \text{ if } |x| < \Delta x/2 \text{ and } |y| < \Delta y/2 \\ 0 \text{ otherwise} \end{cases} \tag{5}$$

For box filtering, the intersection of the one-pixel wide line segment and the pixel concerned has to calculated. Let us realize that maximum three pixels may intersect a pixel rectangle in each column if the slant is between 0 and 45 degrees. Let the vertical distance of the three closest pixels to the center of the line be $r$,$s$ and $t$ respectively, and suppose $s < t \leq r$. By geometric considerations $s, t < 1$, $s + t = 1$ and $r \geq 1$ should also hold. Unfortunately, the areas of intersection, $A_s, A_t$ and $A_r$, depend not only on $r$,$s$ and $t$, but also on the slant of the line segment. This dependence, however, can be rendered unimportant by using approximations $A_s \approx (1 - s)$, $A_t \approx (1 - t)$ and $A_r \approx 0$. Variables $s$ and $t$ are calculated for a line $y = m \cdot x + b$:

$$s = m \cdot x + b - Round(m \cdot x + b) = Error(x) \quad \text{and} \quad t = 1 - s \tag{6}$$

where $Error(x)$ is, in fact, the accuracy of the digital approximation of the line as a function of $x$. The color contribution of the two closest pixels in this pixel column is:

$$I_s = I \cdot (1 - Error(x)), \qquad I_t = I \cdot Error(x) \tag{7}$$

($I$ stands for any color coordinate $R, G$ or $B$.) These formulae are also primary candidates for incremental evaluation, since $I_s$ and $I_t$ are both linear, and if the closest pixel has the same $y$ coordinate for an $x+1$ as for $x$:

$$I_s(x+1) = I_s(x) - I \cdot m, \qquad I_t(x+1) = I_t(x) + I \cdot m \tag{8}$$

If the $y$ coordinate was incremented when stepping from $x$ to $x+1$, then:

$$I_s(x+1) = I_s(x) - I \cdot m + I, \qquad I_t(x+1) = I_t(x) + I \cdot m - I \tag{9}$$

The color computation can be combined with an incremental $y$ coordinate calculation algorithm, such as the Bresenham's line generator:

**AntiAliasedBresenhamLine**$(x_1, y_1, x_2, y_2, I)$
  $\Delta x = x_2 - x_1$; $\Delta y = y_2 - y_1$;
  $E = -2\Delta x$; $dE^+ = 2(\Delta y - \Delta x)$; $dE^- = 2\Delta y$;
  $dI^- = \Delta y / \Delta x \cdot I$; $dI^+ = I - dI^-$; $I_s = I + dI^-$; $I_t = -dI^-$;
  $y = y_1$;
  **for** $x = x_1$ **to** $x_2$ **do**
    **if** $E \leq 0$ **then** $E$ += $dE^-$; $I_s$ -= $dI^-$; $I_t$ += $dI^-$;
    **else** $E$ += $dE^+$; $I_s$ += $dI^+$; $I_t$ -= $dI^+$; $y$++;
    Add Frame Buffer$(x, y, I_s)$;
    Add Frame Buffer$(x, y+1, I_t)$;
  **endfor**

This algorithm assumes that the frame buffer is initialized such that each pixel has the color derived without taking this new line into account, and thus the new contribution can simply be added to it. This is true only if the frame buffer is initialized to the color of a black background and lines do not cross each other. In general cases $I$ must rather be regarded as a weight value determining the portions of the new line color and the color already stored in the frame buffer, which corresponds to the color of objects behind the new line. The program line "Add Frame Buffer$(x, y, I)$" should be replaced by the weighted summation of the calculated colors and the values already stored in the frame buffer for each color coordinate $R, G, B$.

# Example 3: Drawing 3D lines with depth cueing

When extending the graphics system's capabilities to 3D, the problems of visibility and shading must be addressed. Visibility requires the calculation of the $Z$ (depth) coordinate of the points of the object rendered, while shading means the computation of its color. Depth cueing is a simple trick to enhance the appearance of 3D wire-frame images. It uses more intense colors for rendering those pixels which are nearer to the eye position. The basic line drawing algorithm can generate the pixel addresses of a 2D digital line, therefore it must be extended to produce the $Z$ coordinates of the internal pixels and also their color intensities by an incremental algorithm. In order to derive an incremental formula, the increment of the $Z$ coordinate and the intensity $I$ is determined. Let the 3D screen space coordinates of the two end points of the line be $[X_1, Y_1, Z_1]$ and $[X_2, Y_2, Z_2]$ respectively and suppose that the z-buffer can hold values in the range $[0 \ldots Z_{\max}]$. Assume that the intensity factor of depth cueing is $C_{\max}$ for $Z = 0$ and $C_{\min}$ for $Z_{\max}$. The number of pixels composing this digital line is $L = \max\{|X_2 - X_1|, |Y_2 - Y_1|\}$. Since $Z$ varies linearly along the line, the difference of the $Z$ coordinates of two consecutive pixels is $Z = (Z_2 - Z_1)/L$. The perceived color, taking into account the effect of depth cueing, is:

$$I(Z) = I_0 \cdot C(Z) = I_0 \cdot (C_{\max} - \frac{C_{\max} - C_{\min}}{Z_{\max}} \cdot Z) \tag{10}$$

The difference in the color of the two pixel centers is $\delta I = (I(Z_2) - I(Z_1))/L$. Having found fixed point representation for $Z$ and $I$ their calculation can be integrated into the internal loop of the Bresenham's algorithm:

**3D_BresenhamLine** $(X_1, Y_1, Z_1, X_2, Y_2, Z_2, I)$
  Initialize a 2D Bresenham's line generator$(X_1, Y_1, X_2, Y_2)$;
  $L = \max\{|X_2 - X_1|, |Y_2 - Y_1|\}$; $\delta Z = (Z_2 - Z_1)/L$;
  $\delta I = I \cdot ((C_{\min} - C_{\max}) \cdot (Z_2 - Z_1))/(Z_{\max} \cdot L)$;
  $Z = Z_1 + 0.5$;
  $I = I_0 \cdot (C_{\max} - (Z_1 \cdot (C_{\max} - C_{\min}))/Z_{\max}) + 0.5$;
  **for** $X = X_1$ **to** $X_2$ **do**
    Iterate Bresenham's algorithm( $X, Y$ );
    $I \mathrel{+}= \delta I$; $Z \mathrel{+}= \delta Z$; $z = Trunc(Z)$;
    **if** Zbuffer$[X, Y] > z$ **then**
      Write Zbuffer$(X, Y, z)$;
      Write frame buffer$(X, Y, Trunc(I))$;
    **endif**
  **endfor**

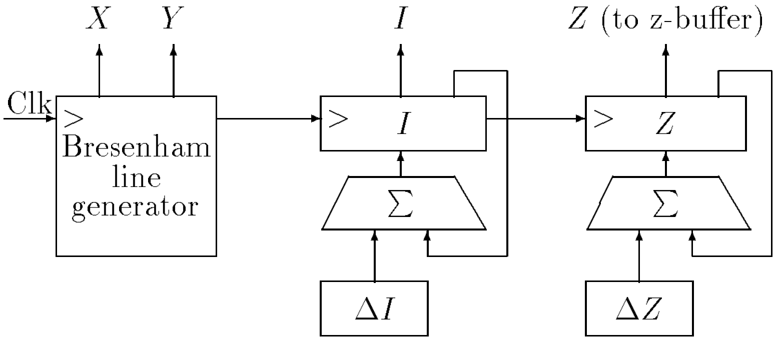


Figure 2: Hardware to draw depth cueing lines

## Multi-variate functions

The results of the previous section can be generalized to higher dimensions, but only the two-variate case has any practical importance, and this can be formulated as follows:

Let a set of two-variate functions be $f_1(k,l), f_2(k,l), \ldots, f_n(k,l)$ and suppose we have to compute them for domain points $S = \{(k,l) \mid l_s \leq l \leq l_e,\ K_s(l) \leq k \leq K_e(l)\}$.

A possible program for this computation is:

**for** $l = l_s$ **to** $l_e$ **do**
  **for** $k = K_s(l)$ **to** $K_e(l)$ **do**
    $F_1 = f_1(k,l)$; $F_2 = f_2(k,l)$; $\ldots F_n = f_n(k,l)$;
    **write**( $k, F_1, F_2, \ldots F_n$ );
  **endfor**
**endfor**

Functions $f$, $K_s$, $K_e$ are assumed to be the truncations of real valued, differentiable functions to integers. Incremental formulae can be derived for these functions relying on Taylor's approximation:

$$f_r(k+1,l) \approx f_r(k,l) + \frac{\partial f_r(k,l)}{\partial k} = f_r(k,l) + \delta f^k(k,l) \tag{11}$$

$$K_s(l+1) \approx K_s(l) + \frac{dK_s(l)}{dl} = K_s(l) + \delta K_s(l),$$
$$K_e(l+1) \approx K_e(l) + \frac{dK_e(l)}{dl} = K_e(l) + \delta K_e(l) \tag{12}$$

The increments of $f_r(k,l)$ along the boundary curve $K_s(l)$ is:

$$f_r(K_s(l+1), l+1) \approx f_r(K_s(l), l) + \frac{df_r(K_s(l), l)}{dl} = f_r(K_s(l), l) + \delta f^{l,s}(l) \tag{13}$$

These equations are used to transform the original program computing $f_i$-s:

$S = K_s(l_s) + 0.5$; $E = K_e(l_s) + 0.5$;
$F_1^s = f_1(K_s(l_s), l_s) + 0.5$; $F_2^s = f_2(K_s(l_s), l_s) + 0.5$; $\ldots F_n^s = f_n(K_s(l_s), l_s) + 0.5$;
**for** $l = l_s$ **to** $l_e$ **do**
  $F_1 = F_1^s$; $F_2 = F_2^s$; $\ldots F_n = F_n^s$;
  **for** $k = Trunc(S)$ **to** $Trunc(E)$ **do**
    **write**( $k, l, Trunc(F_1), Trunc(F_2), \ldots, Trunc(F_n)$ );
    $F_1 \mathrel{+}= \delta f_1^k$; $F_2 \mathrel{+}= \delta f_2^k$; $\ldots F_n \mathrel{+}= \delta f_n^k$;
  **endfor**
  $F_1^s \mathrel{+}= \delta f_1^{l,s}$; $F_2^s \mathrel{+}= \delta f_2^{l,s}$; $\ldots F_n^s \mathrel{+}= \delta f_n^{l,s}$; $S \mathrel{+}= \delta K_s$; $E \mathrel{+}= \delta K_e$;
**endfor**

To realize this transformed program by hardware, a two level hierarchy of interpolators should be built. On the lower level, interpolators have to be allocated for each $F_i$, which are initialized by a respective higher level interpolator generating $F_i^s$. The counters controlling the operation also form a two-level hierarchy. The higher level counter increments two additional interpolators, one for start position $S$, and one for end condition $E$, which, in turn, serve as start and stop control values for the lower level counter. Concerning the length of the temporary registers, note that in the modified algorithm the longest path where the round-off errors can accumulate consists of $\max_l\{l-l_s+K_e(l)-K_s(l)\} \le P_k+P_l$ steps, where $P_k$ and $P_l$ are the size of the domain of $k$ and $l$ respectively. The minimum length of the fractional part can thus be calculated from $b_F > \log_2(P_k+P_l)$.

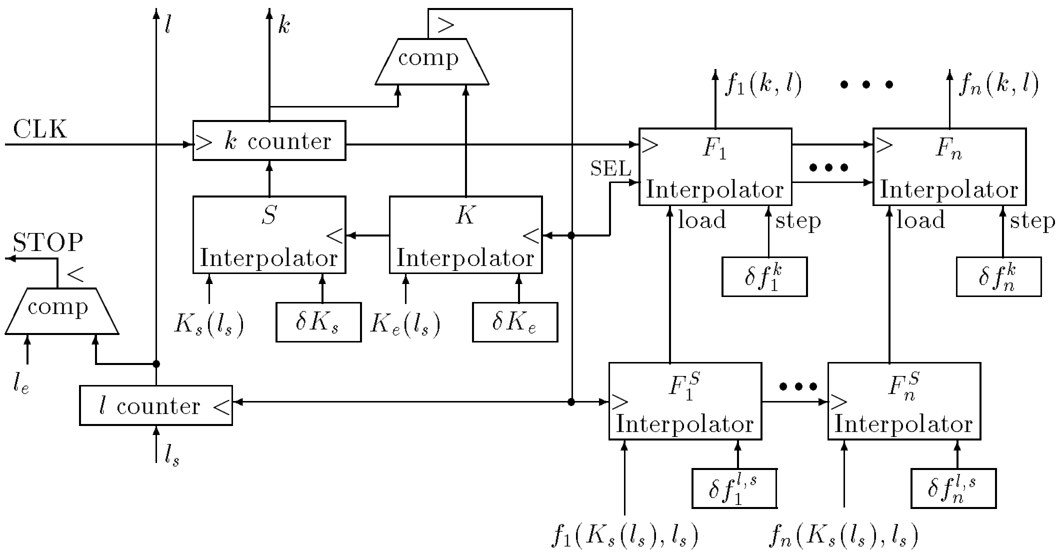


Figure 3: Hardware realization of two-variate functions

# Example 4: Z-buffer, Gouraud shading triangles

For hidden surface elimination, the z-buffer method is often used together with Gouraud shading. When an internal pixel is rendered, its color and its $Z$ coordinate in a so called screen coordinate system are required. If the surface is planar — usually a triangle —, then the $Z$ coordinates of the internal points can be calculated by linear interpolation. Gouraud shading also approximates the color variation by a linear function making the process of color computation similar to that of the calculation of the $Z$ coordinate. Having approximated the surface by a polygon mesh, the surface is given by the set of mesh vertices, which should have been transformed to the screen coordinate system. Without loss of generality, we can assume that the polygon mesh consists of triangles only. Let the vertices of the triangle in screen coordinates be $\vec{r}_1=[X_1,Y_1,Z_1]$, $\vec{r}_2=[X_2,Y_2,Z_2]$ and $\vec{r}_3=[X_3,Y_3,Z_3]$ respectively. The scan conversion algorithms should determine the $X,Y$ pixel addresses and the corresponding $Z$ coordinates of those pixels which belong to this triangle (figure 4). If the $X,Y$ pixel addresses are already available, then the calculation of the corresponding $Z$ coordinate can exploit the fact that the triangle is on a plane, thus the $Z$ coordinate is some linear function of the $X,Y$ coordinates. This linear function can be derived from the equation of the plane. Using the notation $\vec{n}$ and $\vec{r}$ to represent the normal vector and the points of the plane respectively, we get:

$$\vec{n}\cdot\vec{r}=\vec{n}\cdot\vec{r}_1 \quad \text{where} \quad \vec{n}=(\vec{r}_3-\vec{r}_1)\times(\vec{r}_2-\vec{r}_1) \tag{14}$$

Let us denote the constant $\vec{n}\cdot\vec{r}_1$ by $C$, and express the equation in scalar form, substituting the coordinates of the vertices ($\vec{r}=[X,Y,Z(X,Y)]$ ) and the normal of the

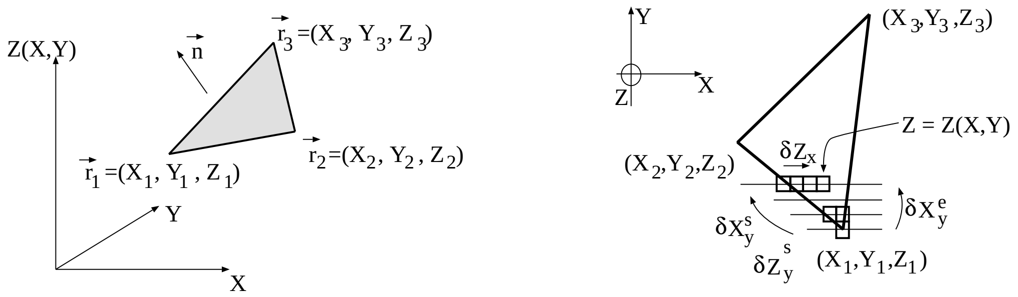


Figure 4: Screen space triangle and the incremental concept

plane ($\vec{n} = [n_X, n_Y, n_Z]$). The function of $Z(X,Y)$ is then:

$$Z(X,Y) = \frac{C - n_X \cdot X - n_Y \cdot Y}{n_Z} \tag{15}$$

This linear function must be evaluated for those pixels which cover the pixel space triangle defined by the vertices $[X_1, Y_1]$, $[X_2, Y_2]$ and $[X_3, Y_3]$. In order to make the boundary curve differentiable and simple to compute, the triangle is split into two parts by a horizontal line at the position of the vertex which is in between the other two vertices in the $Y$ direction. Two different orientations (called left and right orientations respectively) are possible, in addition to the different order of the vertices in the $Y$ direction. Since the different cases require almost similar solutions, we shall discuss only the scan conversion of the lower part of a left oriented triangle, supposing that the $Y$-order of the vertices is: $Y_1 < Y_2 < Y_3$. The computational burden for the evaluation of the linear expression of the $Z$ coordinate and for the calculation of the starting and ending coordinates of the horizontal spans of pixels covering the triangle can be significantly reduced by the incremental concept (figure 4). Expressing $Z(X+1,Y)$ as a function of $Z(X,Y)$, we get:

$$Z(X+1,Y) = Z(X,Y) + \frac{\partial Z(X,Y)}{\partial X} \cdot 1 = Z(X,Y) - \frac{n_X}{n_Z} = Z(X,Y) + \delta Z_X. \tag{16}$$

Since $\delta Z_X$ does not depend on the actual $X, Y$ coordinates, it has to be evaluated once for the polygon. In a scan-line, the calculation of a $Z$ coordinate requires a single addition according to equation 16. Since $Z$ and $X$ vary linearly along the left and right edges of the triangle, equations 11 and 12 result in the following simple expressions in the range $Y_1 \le Y \le Y_2$, denoting the variables $K_s$ and $K_e$ used in the general discussion by $X_{start}$ and $X_{end}$ respectively:

$$X_{start}(Y+1) = X_{start}(Y) + \frac{X_2 - X_1}{Y_2 - Y_1} = X_{start}(Y) + \delta X_Y^s \tag{17}$$

$$X_{end}(Y+1) = X_{end}(Y) + \frac{X_3 - X_1}{Y_3 - Y_1} = X_{end}(Y) + \delta X_Y^e \tag{18}$$

$$Z_{start}(Y+1) = Z_{start}(Y) + \frac{Z_2 - Z_1}{Y_2 - Y_1} = Z_{start}(Y) + \delta Z_Y^s \tag{19}$$

Similar formulae can be established for color coordinates $R, G, B$.

The complete incremental algorithm is then:

$X_{start} = X_1 + 0.5$; $X_{end} = X_1 + 0.5$; $Z_{start} = Z_1 + 0.5$;
$R_{start} = R_1 + 0.5$; $G_{start} = G_1 + 0.5$; $B_{start} = B_1 + 0.5$;
**for** $Y = Y_1$ **to** $Y_2$ **do**
    $R = R_{start}$; $G = G_{start}$; $B = B_{start}$; $Z = Z_{start}$;
    **for** $X = Trunc(X_{start})$ **to** $Trunc(X_{end})$ **do**
        $z = Trunc(Z)$;
        **if** $z <$ Zbuffer$[X, Y]$ **then**
            Write$(X, Y, Trunc(R), Trunc(G), Trunc(B))$;
            Zbuffer$[X, Y] = z$;
        **endif**
        $Z \mathrel{+}= \delta Z_X$;
    **endfor**
    $X_{start} \mathrel{+}= \delta X_Y^s$; $X_{end} \mathrel{+}= \delta X_Y^e$; $Z_{start} \mathrel{+}= \delta Z_Y^s$;
    $R_{start} \mathrel{+}= \delta R_Y^s$; $G_{start} \mathrel{+}= \delta G_Y^s$; $B_{start} \mathrel{+}= \delta B_Y^s$;
**endfor**

Having represented the numbers in a fixed point format, the derivation of the executing hardware for this algorithm is straightforward as shown in figure 5.

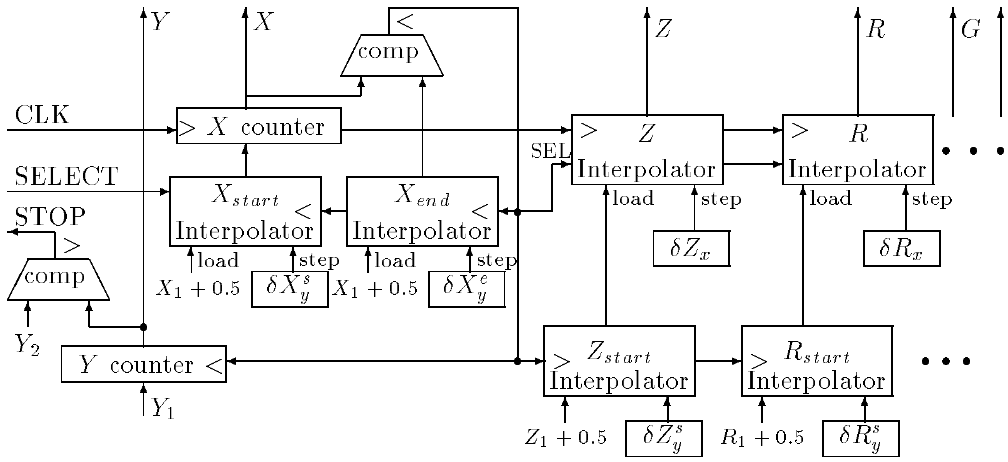


Figure 5: Scan converter for rendering triangles

# References


[Bre65] J.E. Bresenham. Algorithm for computer control of a digital plotter. *IBM Systems Journal*, 4(1):25–30, 1965.

[FvDFH90] J.D. Foley, A. van Dam, S.K. Feiner, and J.F. Hughes. *Computer Graphics: Principles and Practice*. Addison-Wesley, Reading, Mass., 1990.

[SKe95] L. Szirmay-Kalos (editor). *Theory of Three Dimensional Computer Graphics*. Akadémia Kiadó, Budapest, 1995. (to appear).