A VLSI Architecture for Anti-Aliasing

Claudia Romanova and Ulrich Wagner

1. The Aliasing Problem in Computer Graphics

Computer-synthesized images exhibit the typical artifacts of raster displays, called *aliasing, rastering, staircasing* or *the "jaggies"*. Display of an image on a raster CRT requires the sampling the two dimensional image signal $I(x, y)$ to obtain a pixel-based description of intensity. Unfortinately, this sampling process treates the pixel as a mathematical point and the point sampling of an unfiltered object is never correct at any resolution. Aliasing effects (spatial and temporal) are due to undersampling of the image signal. Spatial aliasing occurs when images contain frequencies greater than one half the spatial sampling frequency. Lines that should be straight appear jagged, very small objects may not be visible, portions of long thin objects may disappear.

1.1 Methods for Minimizing Aliasing Effects

There are three basic approaches to remove the aliasing effects:

increase the sampling rate by increasing the display resolution. However, this approach has several limitations, because it also increases the cost of the image production and object edges may fall between pixels at any resolution thus aliasing may occur anyway.

post–filtering: the image is sampled at a higher resolution. Then a digital averaging process is applied to the supersampled image to generate the intensity values in the frame memory. An example is the jagged edges detection and filtering algorithm [6]. The main disadvantage is that details which have been lost during the sampling process cannot been recovered.

pre–filtering or *area sampling* : area sampling treates the image as continuous area of scan lines, where each scan line is a contiguouos band of pixels, each pixel is a square sample, and each sample is of unit edge length. The final pixel intensity is determined by computing the area weighted coverage of all visible objects which cross the pixel. This is equivalent to applying a square area integration filter at a pixel. Visually, this means blurring the picture.

1.2 The Pre–filtering Method

Perfect reconstruction of the original image is still not possible. The pre–filtering algorithms operate by replacing the previous pixel colour value according to the following blending function:

$$C(x,y) \leftarrow \alpha \cdot C_{obj} + (1-\alpha) \cdot C(x,y),$$

where C_{obj} is the colour of the object being drawn, $C(x,y)$ is the previous colour of the pixel (x,y) and α is the response of a low pass filter to the object at (x,y). There are two properties that characterize the pre–filtering algorithm – the filter computation scheme and the filter type. The filter type determines how objects look like. The sampling filter is typically a box, a triangle (tent) or truncated Gaussian. Assigning brightness to each pixel proportional to the fraction of the covered pixel area, is a well established approach. Various implementations have been reported, where the fraction is incrementally calculated [12,21,22,23] or stored in a lookup table indexed by the distance from the object to the pixel [4]. The common property of the above mentioned algorithms is that they are well suitable for vector generation. The other alternative is the approximation of the covered area (α) by using a coverage mask. Each pixel is virtually divided into subpixels, the coverage rate is determined by the number of the subpixels crossed by the object. Section 2 gives a detailed survey of the known anti-aliasing approaches.

1.3 Stochastic Sampling

The last anti-aliasing approach to be mentioned is the stochastic sampling. The image signal is sampled at irregularly placed points and instead of the "jaggies" we get noise. Many feel that this noise is less disturbing for the human observer that the aliasing. The main application area of the stochastic sampling is ray tracing. The investigations in this methods are concentrated on finding good sampling distributions and filtering methods to adaptively increase the sampling rate in regions of the image with high frequencies. A good representation of the stochastic sampling could be found in [9].

1.4 The Role of Gamma Correction

Without accurate gamma correction, anti-aliasing through pixel brightness control is ineffective. Unfortunately, the response of typical video colour monitors is non-linear. Thus, when a linear image is loaded into the frame buffer, a video lookup table is also loaded to correct for the non-linearity of the monitor. The monitor correction function is an exponential function of the form:

$$\text{lookup value} = \text{intensity}^{1.0/\gamma}$$

Gamma(γ) represents the non-linearity of the monitor and usually is in the range 2.0 to 3.0. Another problem is that an increment of 1 in the image intensity is mapped into a much larger increment for the display. Thus, the available resolution of the display system in the lower intensity range is not being used [2]. One solution of the problem is to use a lookup table with logarithmic mapping of 12-bit input to 8-bit output values.

The recalculating of the display-data values by compensation tables [5] is one choice for gamma correction.

2. Review of Existing Anti–Aliasing Techniques

All algorithms mentioned in this section, except the cited subpixel coverage mask methods, were implemented by the authors for a straight line drawn from a pixel centre at a gradient 5/7. It is to point out, that only the using of a coverage mask allows several objects per pixel (more than two). All algortihms can be modified to antialias polygon edges, too.
The task of the anti-aliasing process is to *determine the covered area as exact as possible.* Two main approaches can be distinguished.

2.1 Exact Computing of the Covered Area

The most exact algorithm for calculating the intersected area is the method by *Pixel Integration*, described in [22]. The anti-aliased lines drawn with the Field's algorithms [12] have a width less than one pixel and look on the raster display thinner, but the area crossed by the vectors is determined exactly, too. These methods work incrementally beginning with the vector start point, i.e. they are *object oriented* and take into consideration three pixels per column. Algorithms for drawing anti-aliased objects on raster output devices are presented in great detail in [11].

2.2 Approximate Computing of the Covered Area

All methods determining the intensity independently of the pixel location can be indicated as *pixel oriented.*

2.2.1 Using the Distance

The following methods [16,21] use the fractional part of a coordinate value of the boundary edge position as a criterion for pixel intensity determination. The algorithm of Fujimoto and Iwata [14] calculates the distance between the vector and the pixel. In both approaches the pixel intensity is inversely proportional to the calculated distance. In [23] the control parameter of the Bresenham's algorithm is interpreted as a measure of the distance from the straight line to the pixel centre. This algorithm is known as modified Bresenham's algorithm. These algorithms involve two pixels per column.
The algorithm cited by Gupta and Sproull in [17] uses the perpendicular distance from the pixel centre to the edge as index into a lookup table with precomputed conic filtering function to determine intensities at the pixels, but it requires seperate treatment for the end points of the vector. The above algorithms rely upon incrementally computations, too. Since in our system architecture the same distance is available, this approach could be used in the Filtering Stage, but it has several disadvantages, still to be discussed.

2.2.2 Using the Coverage Mask

The methods presented in [3,13,19] and in this paper compute an anti-aliased image using subpixel mask. The bitmask can be interpreted as a 2D array of subpixels, each

subpixel being off or on depending on the intersected area of the object with the pixel. The quantized area is computed by summing the number of bits that are on in the bit-mask and dividing by the total number of bits in the mask. So, this fractional coverage multiplied by the object intensity determines the final intensity of the displayed pixel. The coverage mask can be either generated or determined by indexing a precomputed table of bitmasks. Our algorithm is similar to the well known "A–buffer" algorithm [20]. The authors of [1] proposed the "gz-buffer". The gz-buffer can be split into an usual z-buffer and a g-buffer containing the geometrical information g for the pixel. Each pixel is divided into sixteen subpixels, therefore the z-buffer needs only four bits per pixel. The approach in the mentioned paper uses the oversampling process for anti-aliasing and in the second one [15] the modified Bresenham's algorithm is applied.
On the Workstation PS 390 of Evans & Sutherland a real-time anti-aliasing technique in hardware is already implemented, which uses 8 x 8 subpixel resolution and produces anti-aliased vectors of high quality.

2.2.3 Compositing Technique

According to this technique the image is described by a quadruple (r, g, b, α), where α is called alpha channel and indicates the covered fraction by the object. The α value lies in the interval [0.0,1.0], where 0.0 corresponds to no coverage and 1.0 to full coverage, the fractions mean partial coverage. The Duff's approach in [10] combines the α compositing algebra with z-buffer to provide simple anti-aliasing.
Assuming a quadratic pixel model the z-values of the object (foreground) at the four pixel corners are computed and compared with these of the previous object (background). Then the total area of both objects is found and the pixel colour values are altered by this value. The (r, g, b, α, z) representation for more than two objects assumes that they are either adjacent in depth or requires knowledge of their front-to-back ordering. The method fails by two elements sharing an edge.

2.3 Error Estimations of Quantized Area Sampling Technique

The difference between the exact and approximated area depends both on the line slope and the chosen subpixel resolution (N). The quantized pixel area (α) can be expressed as:

$$\alpha = \frac{\sum_{i=1}^{N} SubpixelValue_i}{N^2} \qquad SubpixelValue_i \in [0, 1]$$

The worst case corresponds to horizontal and vertical lines, which cross the subpixel centers and the maximum error is equal to $\frac{1}{2N}.100\%$. Statements about errors of other anti-aliasing algorithms are scarce, but when they are given, the errors lie in range of $8 - 12\%$.

3. The PROOF System for Computer Image Generation

The idea of the architecture of our system PROOF (Pipeline for Rendering in an Object Oriented Framework) is based on an object space subdivision approach, given by Cohen [8] and extended by Weinberg [27] and Straßer [26]. Figure 1 shows a block diagramm of the overall system. The geometric processor executes the perspective and geometric transformations, the clipping and the triangulation of the objects in the scene. Each processor is dedicated to an object primitive in the image space. The object processors are organised in pipeline (Object Processor Pipeline). The Look-Up-Tables between the stages are loaded with parameters neccessary for the following Shading and Filtering Stages. The system operates in two modes: *load* and *pixel*.

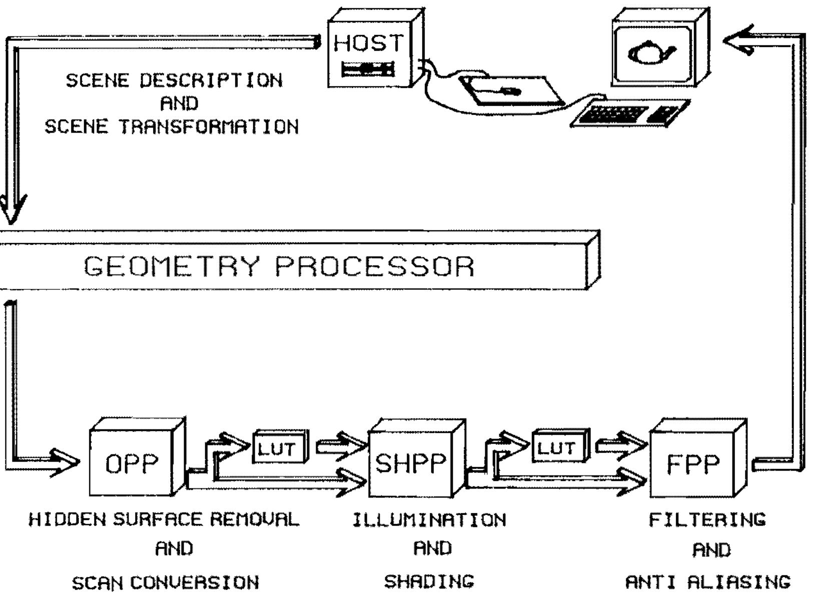
HOST
SCENE DESCRIPTION AND SCENE TRANSFORMATION
GEOMETRY PROCESSOR
OPP
LUT
SHPP
LUT
FPP
HIDDEN SURFACE REMOVAL AND SCAN CONVERSION
ILLUMINATION AND SHADING
FILTERING AND ANTI ALIASING

Figure 1: The Architecture of PROOF

3.1 Load Mode

The same pipeline is used for loading the object data. In the load mode the object processors, the shading processors and the LUT's are preloaded by the geometric processor with specific properties describing the objects and the scene. For example, the first LUT contains parameters needed by the Shading Processor Pipeline such as parameters referring to the position and colour of the light sources in the scene and surface properties of the objects. The second one is loaded with geometrical informations required by the Filtering Stage. Identifying each object from the scene by an object number (object identifier) and using this as address into the LUT's minimizes the bandwidth of the dataflow in the system essentially.

3.2 Scan Conversion and Hidden Surface Removal Algorithms

Pixel computations consist of the scan conversion of the object and the interpolation of the Z coordinates and colour values. The scene to be rendered consists of triangles as objects primitives. A triangle edge is given by its bounding line, described by the

Hessian normal form as follows:

$$g \quad : \quad \cos\theta.x + \sin\theta.y - p = 0$$

The values of $\cos\theta$, $\sin\theta$ and p are determined from the end-points of the line, traversed in an anticlockwise direction, so that the triangle will always lie in the left halfplane of the line.
The scan conversion algorithm applied in the OPP can be classified as *inside testing* algorithm and takes advantage of the method of differences. Every object processor calculates the distance between each edge of its own object and the centre of a pixel (x, y) by the equation:

$$d_i(x,y) \quad = \quad \cos\theta_i.x + \sin\theta_i.y - p_i \tag{1}$$

for $i = (1,2,3)$. Evaluating the value of d_i for every pixel requires two multiplications and two additions. Fortunately, the calculations can be made incrementally with only one addition per step. The value of $d_i(x,y)$ for successive pixels is obtained by initializing a register for each edge with the value of $(-p_i)$ and adding $\cos\theta_i$ for each step along the x axis and adding $\sin\theta_i$ for each step along the y axis. The basic rendering operations are as follows:

command	execution
new frame	$d_i(0,0) \quad = -p_i$
new pixel	$d_i(x+1,y) = d_i(x,y) + \cos\theta_i$
new scan line	$d_i(0,y+1) = d_i(0,y) + \sin\theta_i$

The value of d_i will be negative for all pixels that are on the left side of the triangle edge and positive for all pixels that are on the right side of the triangle edge. If any d_i is in the range $[-R, +R]$ (R denotes the pixel radius and for a quadratic pixel with unit length is equal to $\frac{\sqrt{2}}{2}$), the pixel is partially covered from the object. If all d_i are greater than $+R$, the object covers the pixel totally and if at least one d_i is less than $-R$ and another d_i is (are) greater than $+R$, then the pixel is completely outside the polygon.
In addition to the object coordinates at each vertex of the object, its depth and its colour are stored in the object procesor. It generates point samples of these values for each pixel within the triangle by linear interpolation between the vertex values incrementally in a manner similar to the calculation of the function for the distance (1). The hidden surface removal approach in the OPP relies upon the depth-buffer algorithm, more exact the distributed z-buffer. For each pixel a list with objects is maintained, which cross the pixel. The next sections explain briefly the architectures of each stage of PROOF.

3.2 Object Processor Pipeline

All object processors have the same architecture design, shown in Figure 2, and each object processor consists of primitive processor, comparator, FIFO and control logic. The primitive processor scan converts its object and performs the interpolation of the depth and normal colours vectors. The seven update units work in parallel to achieve high performance. The comparator is responsible for the decision if the interpolated z value of the local object lies further back than the objects' depth values stored in the list. The FIFO receives the data sent from the last object processor and serves as a buffer between the singles object processors. During the scan conversion, each object

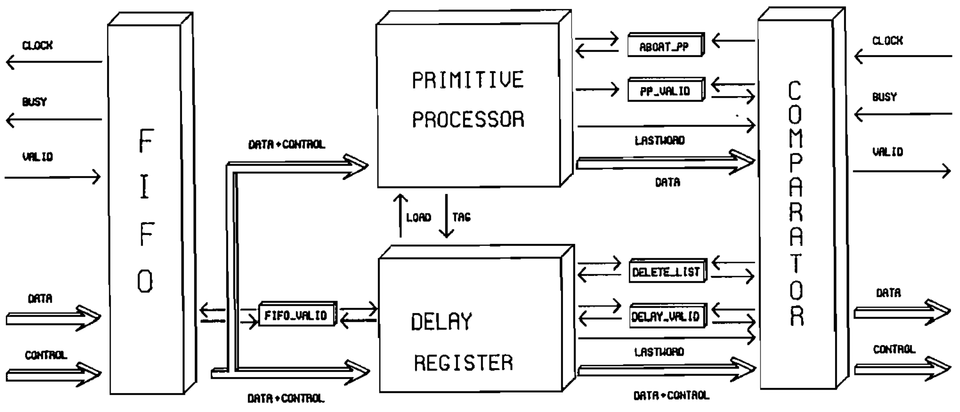

Figure 2: The Internal Design of the Object Processor

processor receives from the previous processor a depth sorted list of potentially visible objects for the current pixel. If the object processor's own object lies within the triangle and the z value is less than that for the triangle computed by preceding object processor for this pixel, the local object is inserted in the list. Else if the object totally hides the previous candidate objects and is non-transparent, the hidden objects are removed from the list. At the end of the object processor pipeline a list of objects for the current pixel emerges.

3.3 Shading Processor Pipeline

Afterwards, these data pass a multiple light source shading processor pipeline (SHPP) in which a Phong–like shading is performed. The illumination model implemented in the SHPP includes terms for ambient light, diffuse and specular relection. The calculations are implemented such, that only a single type processor is needed. The Shading Processors are connected in a series to form a pipeline, too. The pipeline is initialized with the ambient term. For each light source in the scene two shading processors are calculating the other two terms. A review about the investigations in reducing the Phong Shading method is given in [7].

3.4 Filtering and Anti-aliasing Stage

The last stage is the filter processor pipeline (FPP) which for each potentially visible object from the list re-scans it over a high resolution subpixel mask and applies a predetermined filter function over the visible portions of the object. Finally, these sums are multiplied by the object's colour. The end colour sums are sent to the frame buffer.

Subpixel position information is required by Filtering Stage to properly antialias an object. The key elements of our system are described in detail in [24] and [25].

4. Algorithm for Filtering

In order to generate the subpixel coverage mask of an object we need geometric information about object intersection points with the pixel area. The object oriented anti-aliasing algorithms are to be rejected, because we would need a filter processor per object, too. That would be too expensive. The disadvantage of the method based on the distance is that it is not applicable for more than two objects per pixel. The only apparent way to solve this problem is subpixel coverage mask generation.
The algorithm in the Filtering Stage is performed on a per-pixel basis so as to preserve the pixel area coherence. It makes use of the distance calculated in the OPP. Our task is to determine for every object of the list how many subpixels from the object are crossed. We calculate then the perpendicular distances between each subpixel centre and the triangle edges. Since the function $d_i(x, y)$ is linear, we can obtain the values

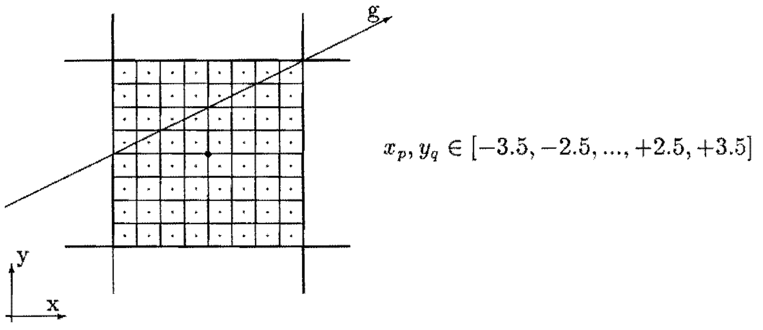

Figure 3: Pixel Area with 8 x 8 Subpixel Resolution

only with adding the first forward differences of the function for n-step change in x- and y-direction. The distance between the triangle edge and the subpixel (p, q) within the pixel area can be expressed as:

$$d_{pq}(x, y) = \cos\theta_i.(x.sp + x_p) + \sin\theta_i.(y.sp + y_q) - p_i, \tag{2}$$

with the following meanings:

x, y	:	x– and y–coordinates of the pixel
sp	:	subpixel resolution
x_p, y_q	:	offsets in x– , y–direction within the pixel area
$\cos\theta_i$, $\sin\theta_i$, p_i	:	edge parameters in Hessian normal form
$d_{pq}(x, y)$	:	distance from subpixel centre (p, q) to the edge

The equation (2) can be simplified to:

$$\begin{aligned} d_{pq}(x, y) &= \underbrace{\cos\theta_i.x.sp + \sin\theta_i.y.sp - p_i}_{d_i(x,y)} + \cos\theta_i.x_p + \sin\theta_i.y_q && (3) \\ &= d_i(x, y) + \cos\theta_i.x_p + \sin\theta_i.y_q && (4) \end{aligned}$$

Thus, the desired distance can be obtained only by adding or substracting appropriate coefficients to the value of $d_i(x,y)$, namely multiples of $\cos\theta_i$ and $\sin\theta_i$. Figure 3 shows the pixel area and the corresponding coefficients. All bits in the coverage mask for these the calculated subpixel distance $d_{pq}(x,y) \leq 0$ are to be set.
The object list is depth sorted with respect to the pixel centre. The assumption, that this list is correctly sorted for each subpixels too, would cause visible errors, as the following example illustrates. If there are two objects in a pixel, one covering the pixel totally, another one, lying behind, but having greater z gradient and not covering the center of the pixel. Since the z-values in the OPP are always computed at the pixel center, it can happen that the interpolated z-value of the second object is smaller than the z-value of the first one. This causes the second object to be visible, in spite of being hidden by the totally covering object. For this reason we need a z-buffer within a pixel region, too. Analogous to the determination of the subpixel distance we can calculate the needed depth values at subpixels as follows:

$$z_{pq}(x,y) \quad = \quad z(x,y) + A_z.x_p + B_z.y_q \tag{5}$$

For transparent objects we suppose, that they are correctly sorted in the list.
Since the PROOF-architecture supports translucency as object property, a transparency modul in the Filtering Stage is integrated too. Generally, assuming that n transparent objects each with colour C_{obj_i} and transparent factor t_i cross a given pixel, the final pixel colour is to be determined by the equation $C_{pixel} = \sum_{i=1}^{n} C_i + C_{bg}$. The bg denotes the background in the scene with $t_{bg} = 0$ and each term C_i is expressed as:

$$\begin{array}{ll}
\text{object colour} & \text{remaining term} \\
C_1 = (1-t_1)\cdot C_{obj\_1} & R_1 = 1 - t_1 \\
C_2 = (1-R_1)\cdot(1-t_2)\cdot C_{obj\_2} & R_2 = R_1 + (1-R_1)\cdot(1-t_2) \\
\vdots & \vdots \\
C_n = (1-R_{n-1})\cdot(1-t_n)\cdot C_{obj\_n} & R_n = R_{n-1} + (1-R_{n-1})\cdot(1-t_n) \\
C_{bg} = (1-R_n)\cdot C_{bg} &
\end{array}$$

Fortunately, only one multiplication depending on the current transparent factor (t_n) and the remaining term (R_{n-1}) is needed. The same approach can be used at subpixel level, as described later. By test and display of different images we found, that 8×8 subpixel resolution in the xy plane and 4×4 subpixel resolution for the depth value seem to be reasonable. The algorithm antialiases images of wire-frames as well as triangles. The algorithm handles adjacent and intersecting polygons thanks to the geometric information given by the object parameters $\cos\theta_i$ and $\sin\theta_i$.

5. Architecture Design of the Filtering Stage

The Filtering and Anti–Aliasing Stage (FAAST) is divided into three distinct processing stages: d–, z–update units (hereafter Update Units) and subpixel processor area (SPA) (Figure 4). The internal hardware structures of the Update Units and SPA are quite different, being designed for their dedicated tasks. Since neither feed-back of data nor that of control from the SPA to the Update Units is required, the operations of the SPA might be completely overlapped in a pipelining manner with that of the Update Units, which are then processing the next object.

5.1 The d– and z–Update Units

Calculating the multiples of $\cos\theta_i$ and $\sin\theta_i$ (A_z and B_z for the depth value) can be done easily with a binary tree of adder. Figure 5 illustrates a binary-tree multiplier, as used in the Pixel Planes System. Each node of the binary tree has two inputs (top and side) and two outputs (left and right), which are bit-parallel and synchronous with the system clock. The value of the left output is the value of the top one delayed by one clock cycle, the right output is the sum of the top and side inputs. The value at the side input depends on the node level in the tree and is multiple of a and a power of 2. The last tree stage delivers the inverted value of the input, too. For a pixel grid of size $N \times N$ subpixels (assuming N to be power of 2), the tree has $n = \log_2(N/2)$ levels and the value of each side input is $2^n.a$, where n-th level corresponds to the top node of the tree. The total number of the required additions in the tree is $\sum_{m=1}^{n} 2^{n-m}$.

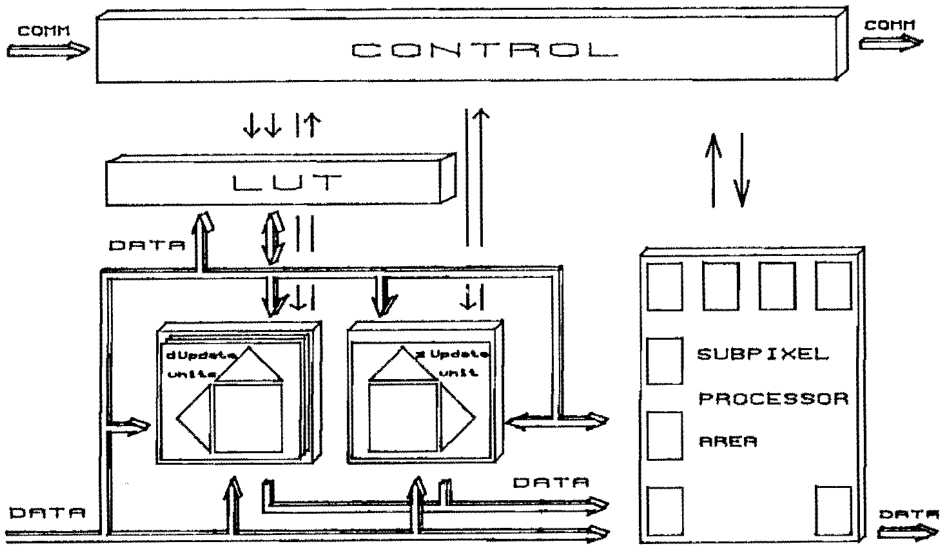

Figure 4: The Architecture of FAAST

The Update Units are similar in their architecture and the task of them is the generation of the subpixel coverage mask. The d-Update Units take advantage of the distance calculated in the OPP, as described earlier, and using the edge parameters loaded in the LUT calculate for each subpixel of the pixel area the distance from the subpixel centre to the object edge. So the 2-dimensional subpixel coverage mask in xy plane can be generated for each object in the sorted list. The task of the z-update unit is similar, but we have to determine the z-values within the pixel area, here. Each type Update Unit, whose block diagramm is shown in Figure 6, is built from two trees and an array of identical Processing Elements (PEs). For a $N \times N$ subpixel resolution we need N^2 PEs for each Update Unit. A PE itself is composed of (de)multiplexing devices, registers

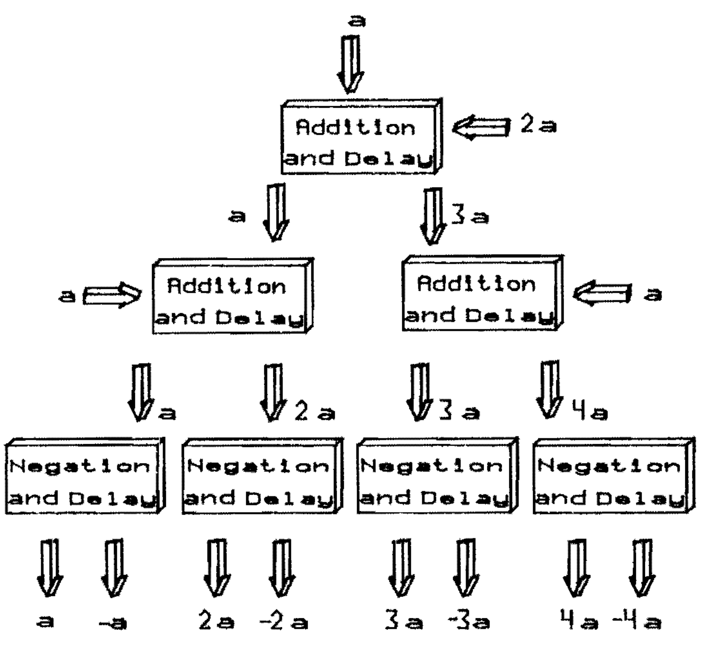

Figure 5: The Binary-Tree Multiplier

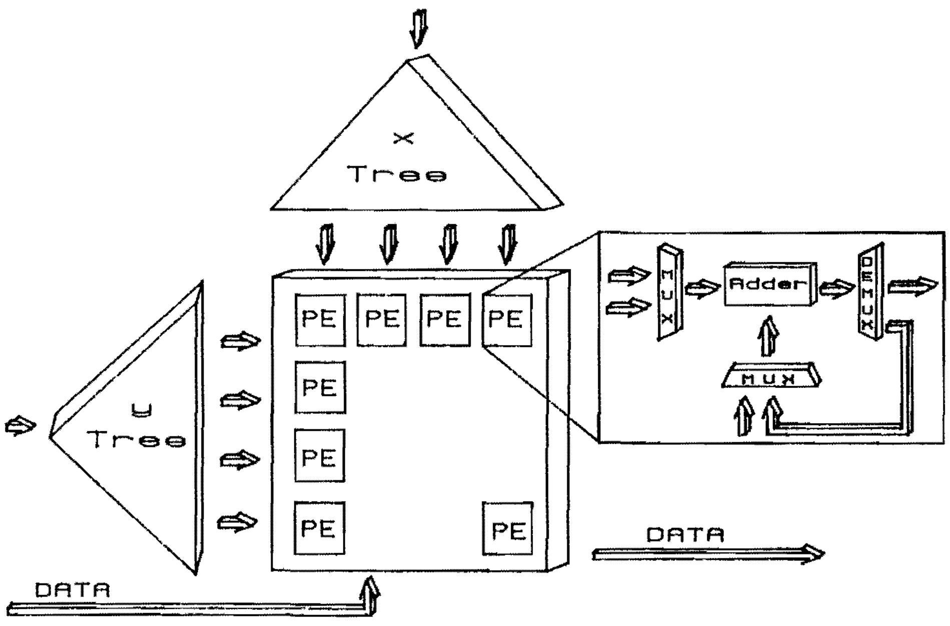

Figure 6: The Update Unit

and an adder. The structures of the trees in the Update Units differ in the number of needed adders and in the bit-width of the addition. The multiplier tree takes in an N-bit data and outputs $(N+3)$-bit-long result. The following table shows the difference between the trees in both types of Update Units, assuming that all adders in the tree have the same bit width:

tree in	number of		bits of	
	levels	adders	input	adder
d-update unit	2	3	16	19
z-update unit	1	1	36	39

The output of the d-Update Units is the 64 bit subpixel coverage mask resulting from the logical AND of the coverage masks for each edge of the triangle. And the results of the z–Update Unit are the depth values for 4×4 subpixel resolution within the pixel area.

5.2 The Subpixel Processor Area

The SPA operates as a single instruction, multiple data (SIMD) processor, executing each instructions on all subpixel processors at the same time. Sixteen Subpixel Processors (Figure 7) constitute the Subpixel Processor Area and each of them consists of registers for object colour, for the transparent factor and for the object depth value at the subpixel and four Subpixel Processor Units (Figure 8). The outputs of the SPU are the colour components for all objects at the current subpixel (sp_rgb_out), which are processed through two stages of adders. Additional summations are made outside of the SPP in a sequence of four adders stages. For each covered subpixel of the object mask the input z-value of the object at that point is compared to the value stored in the z-register. If the comparison is successful, the three colour component values of the object are copied into the colour registers and the z-register is updated with the input z-value. At the end of the procedure all colour components for the actually pixel are summed, divided by the total number of the subpixels and sent to the frame buffer. The algorithm in the Filtering and Anti-aliasing Stage can be described by the following C-like functions:

```
/*               anti-aliasing algorithm in FAAST                   */
/*                                                                  */
  for each pixel(x,y) do
    { null sp_rgb_out, final_rgb;
      initilize sp_z;
      for every object from object_list do
       { null obj_mask, sp_rgb;
         determine obj_mask and obj_z;
         spu(); }
      add all sp_rgb_out to final_rgb; }
/*                                                                  */
/*    spu realizes the tasks of the subpixel processor unit         */
/*                                                                  */
spu()
{
 if (obj transp) and (all prev obj transp) and (subpixel set) do
```

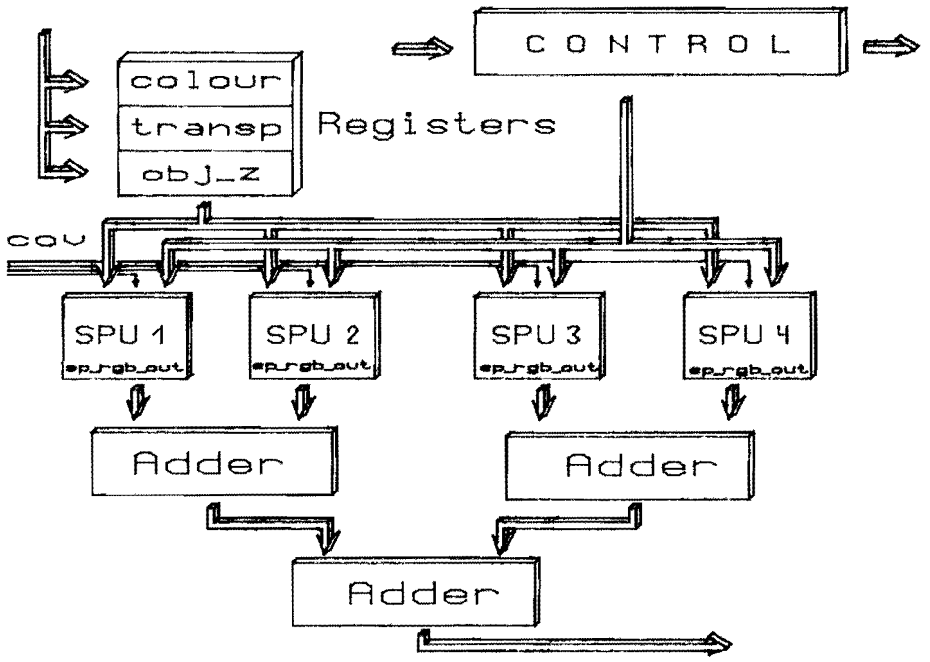

Figure 7: The Structure of the Subpixel Processor

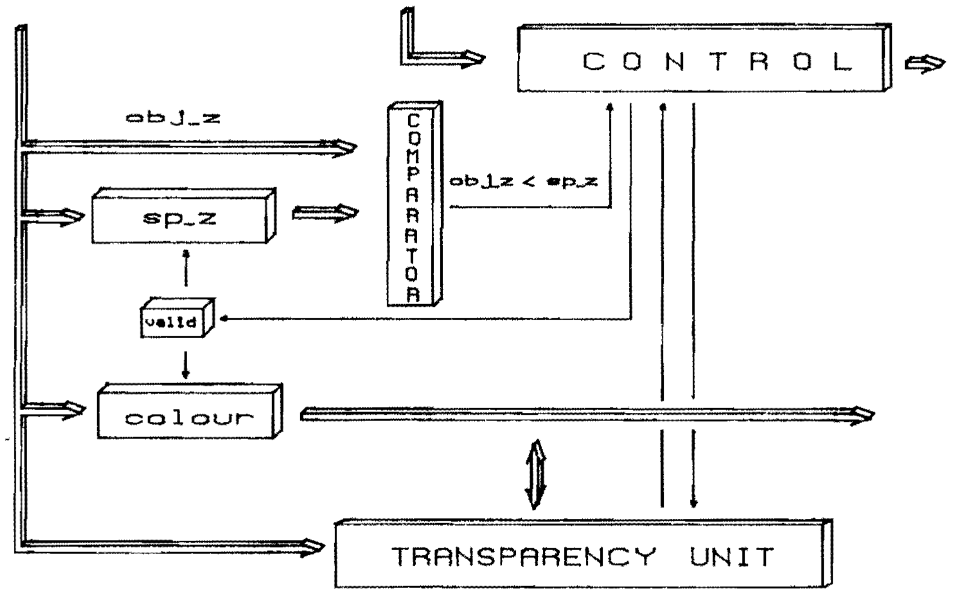

Figure 8: The Design of the Subpixel Processor Unit

```
   { activate transp_unit; }
  if (obj non transp) and (subpixel set) and (obj_z < sp_z) do
   { move obj_rgb to sp_rgb;
      move obj_z   to sp_z; }
  if (last obj) and (subpixel not set) do
   { move obj_rgb to sp_rgb;
     determine sp_rgb_out; }
}
```

5.3 The Lookup Table and Control Unit

The parallel processing manner of the Update Units allows, that the LUT's can be divided into two parts and each part is connected to the Unit by data bus with defined bit width. All Control Units can be realized as PLAs.

6. System Performance Estimation

The pixel processing time is limited by the longest updating time of any stage in the whole pipeline. We restrict our considerations to the processing time in the FAAST. The Update Units process the objects from the list successively and the SPA neede data for all objects in order to calculate the final pixel colour components. The processing time of the Update Units is more decisive than that of the the SPA. Supposing that there are three objects in the list for the current pixel, the Update Units have to determine for two objects (the third one is the background and covers the pixel completely) the subpixel coverage mask, i.e. $t_{pixel} \leq 2.t_{UpdateUnit}$. The processing in the Update Unit includes:

activity	operation	number of needed clock cycles
fetching the object parameters	access to the LUT	1
calculating the coefficients	2 additions and negation	3
determine the d_{pq}	2 additions and evaluate the sign of d_{pq}	3

The main operation to be performed in the Update Unit is addition. The carry–skip adder presented in [18] is well suited for addition of data with large width, e.g. the addition of two 20-bit binary numbers is performed in $10T$ (T is the gate delay). The negation operation needes no extra time, because it can be included in the following addition by setting the input carry to one. So, assuming that the T is $2ns$ the total processing time of an Update Unit is approximately $150ns$ and the $t_{pixel} = 300ns$.

7. Conclusions

Anti-aliasing is needed to preserve high quality images. The described algorithm and architecture to solve this problem has been dictated by the PROOF architecture. The proposed solution seems to be realisable with today's technology. It demonstrates, that anti-aliasing in real time system is feasible.

References

[1] Beigbeder, M., Ghazanfarpour, D., Peroche, B.: The "GZ-Buffer" Method for Antialiasing. International Electronic Image Week, April 1986.

[2] Blinn, J.F.: Dirty Pixels. IEEE Computer Graphics and Applications, 100–105, July 1989. Jim Blinn's Corner.

[3] Carpenter, L.: The A-buffer, an Antialiased Hidden Surface Method. Computer Graphics, 18(3):103–108, July 1984.

[4] Catmull, E.: An Analytic Visible Surface Algorithm for Independent Pixel Processing. Computer Graphics, 18(3):109–115, July 1984.

[5] Catmull, E.: A Tutorial on Compensation Tables. Computer Graphics, 1–7, 1979.

[6] Chryssafis, A.: Anti-Aliasing of Computer – Generated Images: A Picture Independent Approach. Computer Graphics Forum, 5:125–129, June 1986.

[7] Claussen, U.: On Reducing the Phong Shading Method. In F.R.A. Hopgood and W. Straßer, editors, EUROGRAPHICS'89, pages 333–344, Eurographics Association, Elsevier Science Publishers B.V. (North-Holland), 1989.

[8] Cohen, D.: A VLSI Approach to the CIG Problem. 1980. Presentation at SIGGRAPH 1980.

[9] Dippe, M.A.Z.: Anti-Aliasing through Stochastic Sampling. Computer Graphics, 19(3):69–78, July 1985.

[10] Duff, T.: Compositing 3-D Rendered Images. Computer Graphics, 19(3):41–44, July 1985.

[11] Field, D.: Algorithms for Drawing Simple Geometric Objects on Raster Devices. PhD thesis, Princeton University, Juni 1983.

[12] Field, D.: Two Algorithms for Drawing Anti-Aliased Lines. Graphics Interface, 87–95, 1984.

[13] Fuime, E., Fournier, Al.: A Parallel Scan Conversion Algorithm with Anti-Aliasing for a General-Purpose Ultracomputer. Computer Graphics, 17(3):141–150, July 1983.

[14] Fujimoto, A., Iwata, K.: Jag Free Images on a Raster CRT. In Computer Graphics, Theory and Applications, pages 2–15, Springer-Verlag, Tokyo Berlin Heidelberg New York, 1983.

[15] Ghazanfarpour, D., Peroche, B.: A Fast Anti-Aliasing Method with a Z-Buffer. In G. Marechal, editor, EUROGRAPHICS'87, pages 503–512, Eurographics Association, Elsevier Science Publishers B.V. (North-Holland), 1987.

[16] Guangnan, N., Tanner, P., Wein, P., Bechthold, Gr.: An Algorithm for Generating Anti-Aliased Polygons for 3-D Applications. Graphics Interface'83, 23–32, 1983.

[17] Gupta, S., Sproull, R.F.: Filtering Edges for Gray-Scale Displays. Computer Graphics, 15(3):1–5, August 1981.

[18] Guyot, A., Hochet, B., Muller, J.M.: A Way to Build Efficient Carry-Skip Adders. IEEE Transactions on Computers, C-36(10):1144–1152, October 1987.

[19] Hoffert, E.M., Bishop, G.: Exact and Efficient Area Sampling Techniques for Spatial Antialiasing. December 1985. Technical Memorandum, AT & T Bell Laboratories.

[20] Kedar, At.: Enhancement and Implementation of the A-buffer Rendering Algorithm. Master's thesis, University of Regina, Department of Computer Science, Regina, Saskatchewan S4S OA4, Canada, February 1987.

[21] Ketcham, R.L.: A High-Speed Algorithm For Generating Anti-Aliased Lines. Proceedings of the SID, 26(4):329–336, 1985.

[22] Pitteway, M.L.V., Olive, P.M.: Filtering Edges by Pixel Integration. Computer Graphics Forum, 4:111–116, 1985.

[23] Pitteway, M.L.V., Watkinson, D.J.: Bresenham's Algorithm with Grey Scale. Communications of the ACM, 23(11):625–626, November 1980.

[24] Schneider, B.-O.: A Processor for an Object-Oriented Rendering System. Computer Graphics Forum, 7:301–310, 1988.

[25] Schneider, B.-O., Claussen, U.: PROOF: An Architecture for Rendering in Object Space. In A.A.M. Kujik, editor, Advances in Graphics Hardware III, Eurographics, Spinger, Berlin, to appear in 1989

[26] Straßer, W.: A VLSI-oriented Architecture for Parallel Processing Image Generation. In G.L. Rejins and M.H. Barton, editors, Highly Parallel Computers, pages 247–258, Elsevier Science Publishers B.V. (North-Holland), 1987.

[27] Weinberg, R.: Parallel Processing Image Synthesis and Anti-Aliasing. Computer Graphics, 15(3):55–61, August 1981.