Efficient Hardware for Antialiasing Coverage Mask Generation

D. Crisu, S.D. Cotofana, S. Vassiliadis
Computer Engineering Laboratory
Delft University of Technology
Mekelweg 4, 2600 GA Delft, The Netherlands
E-mail: {dan, sorin, stamatis}@ce.et.tudelft.nl

P. Liuha
Nokia Research Center
Visiokatu-1, SF-33720
Tampere, Finland
E-mail: petri.liuha@nokia.com

Abstract

An efficient low-cost, low-power hardware implementation of a novel run-time pixel coverage mask generation algorithm for embedded 3-D graphics antialiasing purposes is presented. The proposed algorithm can be incorporated in any antialiasing scheme with pre-filtering that is based on algebraic representation of primitive's edges. When compared with the state of the art, the described algorithm reduces several times the size of the required hardware implementation due to the utilization of the quadrant symmetry property allowing the storage of only the coverage mask information for a few representative edges in one of the quadrants of the plane, the rest of the information being derived on the fly via computationally inexpensive operations.

1. Introduction

With the recent proliferation of increasing powerful mobile platforms for computing and communications, the request for fast, graphics-reach, user-friendly interfaces and entertainment environments opened new market opportunities for 3-D real-time rendering graphics systems meant to accelerate these features. The challenge posed by the formidable cost constraints on products for the mobile consumer market requires a new breed of graphics rendering hardware with very low power consumption and implementation costs which precludes the utilization of the advanced features and the high throughput achieved in high-end systems. However it is mandatory to implement at least, given their reduced physical size raster displays and the prospects of sluggish resolution improvement (less than 10% a year according to [3]), antialiasing hardware to prevent the common problem of the jagged appearance of lines and polygonal edges, among other aliasing artifacts.

In this paper, an efficient low-cost, low-power hardware implementation of a novel run-time pixel coverage mask generation algorithm for embedded 3-D graphics antialiasing purposes is presented. The algorithm is exploiting the quadrant symmetry property allowing the storage of only the coverage mask information for a few representative edges in one of the quadrants of the plane, the rest of the information being derived on the fly via computationally inexpensive operations. The algorithm is presented assuming 4×4 subpixel coverage masks and two's complement number arithmetic, however it has a higher degree of generality: it can be incorporated in any antialiasing scheme with pre-filtering that is based on algebraic representation of primitive's edges, it is independent of the underlying number representation, and it can be adapted to other coverage mask subpixel resolutions with the only prerequisite for the masks to be square. Assuming a 0.18μm IC manufacturing technology, hardware synthesis results of the coverage mask generation circuitry are indicating that our algorithm requires an area of $12270\mu\mathrm{m}^2$ and has a latency of 2.49ns instead of $270375\mu\mathrm{m}^2$ and 4.2ns required by state of the art solutions [8].

The rest of the paper is organized as follows. Background and preliminaries regarding antialiasing with prefiltering are discussed in Section 2. In Section 3, the antialiasing coverage mask generation algorithm is introduced and highlights of its hardware implementation are discussed. Hardware synthesis results and rendered images using accurate SystemC RTL modeling are presented in Section 4. Finally, in Section 5, the conclusions are drawn.

2. Background and preliminaries

Antialiasing schemes can be classified in pre- and postfiltering methods [4]. The algorithm we propose is employed in an antialiasing scheme based on pre-filtering.

Whithin this last category, one efficient approach for triangle rasterization and triangle antialiasing is based on the algebraic representation of triangle's edges with edge functions [5, 7] and normalized edge functions [8]. In hardware implementations for antialiasing with normalized edge functions, subpixel representations of the pixel coverage coded in coverage masks (depicted in Figure 1(a)) are pre-

Proceedings of the Computer Graphics International (CGI'04)
1530-1052/04 $20.00 © 2004 **IEEE**

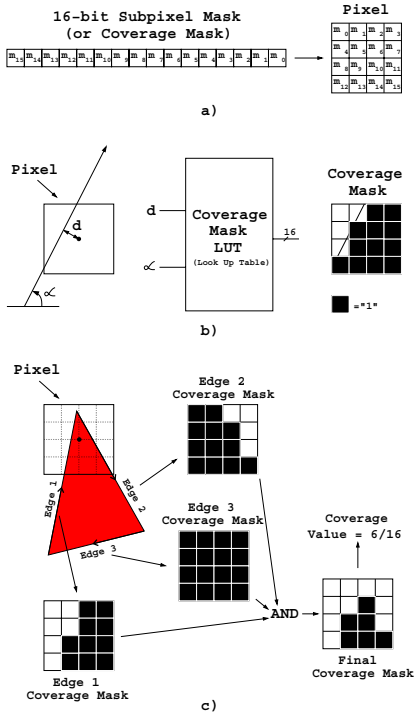

Figure 1. The operating principle of the antialiasing algorithm with normalized edge functions.

computed for various distances to the pixel center and angles of the edge and stored in a coverage mask lookup table (LUT). During rasterization the normalized edge function computed for the current rasterization position (x_M, y_M) is interpreted as a distance and together with the edge slope is used as a LUT address to fetch the coverage mask of the current rasterization position for that edge (presented in Figure 1(b)). The table lookup is performed for all the three edges and the three resultant coverage masks are logically combined to produce a final coverage mask of the triangle over the current rasterization position (x_M, y_M). Then the coverage mask is either employed as in [1] or used to compute a coverage value — the fraction of the pixel covered by the triangle — from the number of lit subpixels out of the total number of subpixels (see Figure 1(c)). Further the coverage value is used to modulate the color (the transparency or the alpha value) which is also computed by interpolation for the current rasterization position (x_M, y_M).

The algorithm we propose can work in conjunction with various normalized edge functions from which the distance d from the pixel center and the angle α with the horizontal can be inferred from parameters of the normalized edge function, it is independent of the underlying number representation, and it can be adapted to other coverage mask subpixel resolutions with the only prerequisite for the masks to be square. For illustrative purposes, the hardware circuits presented in figures are employing two's complement arithmetic. For numerical bit-strings we utilize the following notation: $a^{\{\mathrm{sgn},n,...,0,-1,...,-m\}}$ or $a^{(\mathrm{sgn})}a^{(n)}...a^{(0)}a^{(-1)}...a^{(-m)}$ represents the fixed-point number whose value is $-a^{(\mathrm{sgn})}2^{n+1} + a^{(n)}2^n + ... + a^{(0)} + a^{(-1)}2^{-1} + ...a^{(-m)}2^{-m}$. The bit ranges presented in figures reflect the precision required in the antialiasing datapath of an embedded QVGA graphics accelerator. Thus the numerical ranges for the antialiasing operands used throughout in the paper will be $d_{L_1}^{\{\mathrm{sgn},10,...,-24\}}$, $de_x^{\{\mathrm{sgn},0,...,-20\}}$, $de_x^{\{\mathrm{sgn},0,...,-20\}}$, $\Delta x^{\{\mathrm{sgn},8,...,-4\}}$, and $\Delta y^{\{\mathrm{sgn},8,...,-4\}}$. Also for illustrative purposes, the particular case of 4×4-subpixel coverage masks and a normalized edge function expressed by the L_1-norm distance [8] is selected. The L_1 norm distance can be expressed in relation with the pixel center M as:

$$\begin{aligned} d_{L_1}(M) &= \tfrac{E(x_M,y_M)}{|\Delta x|+|\Delta y|} \\ &= (x_M - x_A)\cdot\tfrac{\Delta y}{|\Delta x|+|\Delta y|} - (y_M - y_A)\cdot\tfrac{\Delta x}{|\Delta x|+|\Delta y|} \\ &= (x_M - x_A)\cdot de_x(\alpha) - (y_M - y_A)\cdot de_y(\alpha) \\ &= \mathrm{sgn}\,(d_{L_1}(M))\cdot d\cdot\tfrac{\sqrt{\Delta x^2+\Delta y^2}}{|\Delta x|+|\Delta y|} \\ &= \mathrm{sgn}\,(d_{L_1}(M))\cdot f(d,\alpha) \end{aligned} \tag{1}$$

The L_1-norm distance $d_{L_1}(M)$ and the parameters $de_x(\alpha)$ and $de_y(\alpha)$ are functions of the Euclidean distance d and the angle α presented in Figure 1(b). Actually any edge vector in the 2D space at any Euclidean distance d from the pixel center M and at any angle α can be identified unambiguously using $d_{L_1}(M)$, $de_x(\alpha)$, and $\mathrm{sgn}\,(de_y(\alpha))$. Therefore, because the angle α is difficult to compute per se, the index used to fetch the coverage masks from the coverage mask LUT can be composed from $d_{L_1}(M)$, $de_x(\alpha)$, and $\mathrm{sgn}\,(de_y(\alpha))$. For practical interest, only the coverage masks for partially covered pixels have to be stored in the coverage mask LUT imposing the range for the L_1-norm distance to be $d_{L_1}(M) \in (-0.5, +0.5)$. Outside this range, the pixel is totally covered or totally uncovered by the triangle edge and the coverage mask can be assigned implicitly to be with all subpixels set or unset depending on the sign of L_1-norm distance $d_{L_1}(M)$. This scheme

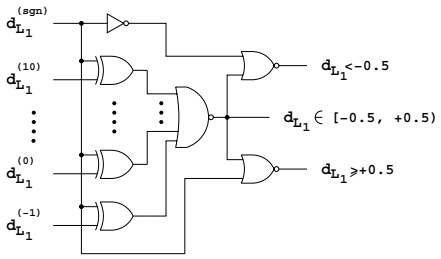

Figure 2. Efficient $\mathrm{d}_{\mathrm{L}_1}$ range detector.

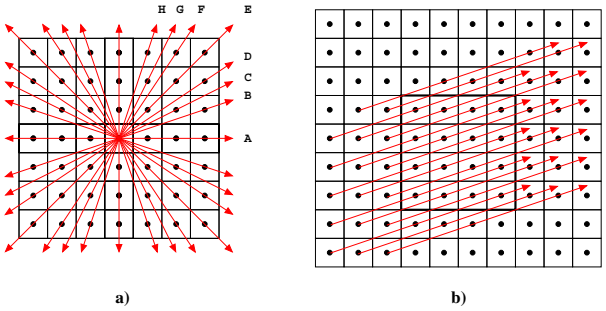

Figure 3. The edge vectors stored in the coverage masks LUT (the subpixels are represented as dotted squares).

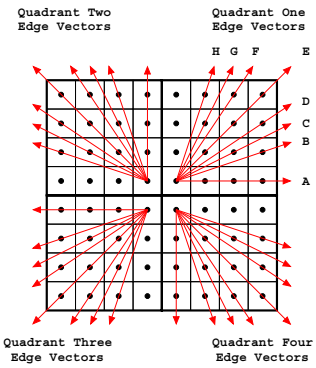

Figure 4. The new method of edge vector class clustering in the four quadrants of the plane (for clarity the edge vectors were drawn in four distinct pixels).

can be easily implemented in hardware with trivial multiplexing circuitry using the two's complement circuit we propose in Figure 2. The ranges for the other parameters that depend on the angle α of the edge vector alone are: $de_x(\alpha) \in [-1, +1]$, and $\text{sgn}\,(de_y(\alpha)) \in \{-1, +1\}$. To keep the coverage masks LUT within reasonable size, the edge vectors can be grouped in edge vector classes and only several representative classes are stored in the coverage masks LUT. *An edge vector class is defined as a set of all the edge vectors with the same* $de_x(\alpha)$ *and* $\text{sgn}\,(de_y(\alpha))$ *values, but with distinct* $d_{L_1}(M)$ *values (the values lie in the above ranges)*. Hence, an edge vector class contains all the edge vectors with the same slope that partially cover a pixel. In the particular case of the EASA antialiasing scheme [8], only 32 edge vector classes from all the four quadrants of plane were stored as presented in Figure 3(a). The 32 edge vector classes were chosen by drawing all the possible edge vectors that were passing through the subpixel centers of a pixel (the edge vectors belonging to edge vector class B are depicted in Figure 3(b)). Then the coverage mask that was stored corresponding to the index given by a combination of $d_{L_1}(M)$, $de_x(\alpha)$, and $\text{sgn}\,(de_y(\alpha))$ was computed by insuring that the number of subpixels lit in the coverage mask was correct plus or minus $1/2$ a subpixel, based on the exact covered area of the pixel. However, additional redundancy had to be incorporated in the LUT to ensure that for two adjacent triangles, both front-facing or both back-facing, a total coverage of more than 1 pixel was impossible and to counteract in two's complement number system the biasing of a rounding scheme based on truncation towards $-\infty$. This increased the coverage masks LUT size to 8k words of 16 bits (8k coverage masks) [8].

3. Proposed coverage mask generation scheme

By maintaining the same system parameters described in the previous section, we propose an algorithm that makes possible a reduction of coverage mask LUT size to no more than 256 16-bit coverage masks, without downgrading the antialiasing quality. Considering that a triangle's oriented edge can be represented as a vector from the source vertex to the sink vertex, the 32 edge vector classes can be clustered according to the quadrant they belong (for horizontal/vertical edge vector classes a convention is made) as presented in Figure 4. Our algorithm proceeds from the consideration that the coverage masks required for the edge vectors of each of the four quadrants correspond to each other in a *rotationally symmetric* manner. That is, if an edge vector, which belongs by its orientation to one quadrant and which requires a specific coverage mask, is rotated in steps of $90°$, the resulting edge vectors in the other quadrants will require the same specific coverage mask rotated in corresponding steps of $90°$. *It is therefore proposed to store only coverage masks for edge vectors belonging by their orientation to a selected one of the quadrants, e.g., to the first quadrant*. The coverage masks for edge vectors belonging by their orientation to another quadrant are obtained by a simple transformation of a coverage mask fetched for a corresponding edge vector belonging to the selected quadrant. Transposing the original edge vector into the selected quadrant and transforming the fetched coverage mask to the quadrant of the original edge vector can be achieved in hardware by computationally inexpensive operations such as simple mask bitwise negations (an inverter per bit of coverage mask), mirrorings, and/or rotations with $90°$ (involving only the proper routing of signals representing the bits in the coverage mask).

The proposed algorithm for coverage mask generation for an edge vector that presents a partial coverage over the current rasterization position (as depicted in Figure 1(b)) is presented in the followings. For a correctness proof of the

algorithm the reader is referred to [2].

Algorithm

1. *Compute* de_x, de_y *for the edge vector and determine the initial quadrant for the edge vector (performed only once per edge);*
2. *Compute* d_{L_1} *for the current rasterization position that the edge touches;*
3. *Quadrant Disambiguation — perform the next operations if the initial quadrant for the edge vector is the following:*
   - *Q1:* $de_x^{Q1} = de_x; \quad d_{L_1}^{LUT\_index} = d_{L_1}$
   - *Q2:* $de_x^{Q1} = -de_y; \quad d_{L_1}^{LUT\_index} = d_{L_1}$
   - *Q3:* $de_x^{Q1} = -de_x; \quad d_{L_1}^{LUT\_index} = -d_{L_1}$
   - *Q4:* $de_x^{Q1} = de_y; \quad d_{L_1}^{LUT\_index} = -d_{L_1}$
4. *Edge Vector Class Disambiguation — Disambiguate the value for* de_x^{Q1} *using bisectors according to Table 1 thus producing a 3-bit* $de_x^{LUT\_index}$ *value, if this disambiguation has produced a wrap-around set the* wrap *flag, else unset* wrap*;*
5. *Use 3-bit* $de_x^{LUT\_index}$ *value and 5 most signifi cant bits of* $d_{L_1}^{LUT\_index}$ *to compose the address and fetch the coverage mask* Mask *from the coverage masks LUT;*
6. *Adjust if necessary the coverage mask* Mask *by producing an intermediary coverage mask* Adjusted_Mask*:*
   - *if* wrap *was set then perform:*
     $\text{Adjusted\_Mask} = \updownarrow (\text{Mask} \circlearrowleft 90^\circ)$
   - *else perform:*
     $\text{Adjusted\_Mask} = \text{Mask}$
7. *If the initial quadrant for the edge vector was the following then compute another intermediary coverage mask* Coverage_Mask*:*
   - *Q1:* $\text{Coverage\_Mask} = \text{Adjusted\_Mask}$
   - *Q2:* $\text{Coverage\_Mask} = \text{Adjusted\_Mask} \circlearrowleft 90^\circ$
   - *Q3:* $\text{Coverage\_Mask} = \textit{not}\ (\text{Adjusted\_Mask})$
   - *Q4:* $\text{Coverage\_Mask} = \textit{not}\ (\text{Adjusted\_Mask} \circlearrowleft 90^\circ)$
8. *Compute the fi nal coverage mask for the edge vector by testing the orientation of the triangle's edges:*
   - *if triangle's edges are oriented clockwise* $(d_{L_1}^{AB}(x_C, y_C) > 0$ *or* $E_{AB}(x_C, y_C) > 0)$ *perform:*
     $\text{Final\_Coverage\_Mask} = \text{Coverage\_Mask}$
   - *else*
     $\text{Final\_Coverage\_Mask} = \textit{not}\ (\text{Coverage\_Mask})$

□

In the description of the algorithm, the operator $\circlearrowleft 90^\circ$ denotes a counter-clockwise rotation with 90° of the 4×4 grid of subpixels that is encoded as a 16-bit coverage mask, the operator $\updownarrow$ signifies a vertical mirroring of the 4×4 grid of subpixels, and the operator not() signifies a bitwise negation of the 16-bit coverage mask.

Due to the fact that the coverage mask LUT contains only instances of the quadrant one edge vector classes the

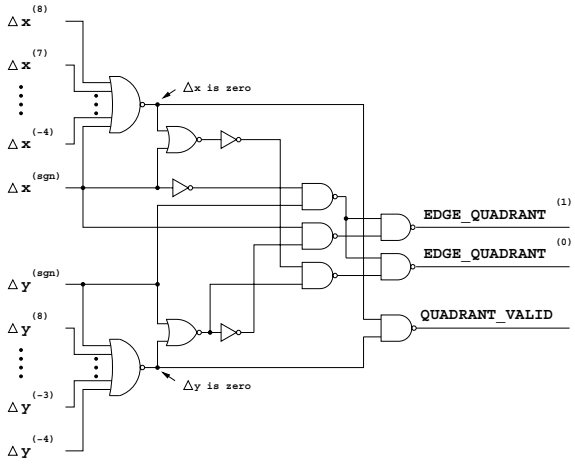

Figure 5. Edge vector quadrant computation.

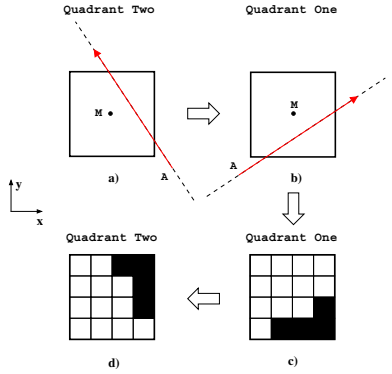

Figure 6. Q2 edge vector coverage mask generation.

indexing scheme became simpler when compared with previous implementations [8]: the index has to be composed taking into account only the transformed $d_{L_1}^{LUT\_index}(M)$ and $de_x^{Q1}(\alpha)$. Now the range for $de_x^{Q1}(\alpha) \in [0, +1)$ (the vertical edge vector class found at the intersection between quadrant one and two belongs according to the convention made to quadrant two) and the quadrant one edge vector classes can be distinguished from each other by the $de_x^{Q1}(\alpha)$ value only.

The algorithmic steps that are particular to the proposed algorithm are explained in the following and their implications for the hardware implementation are also discussed. The **steps 1, 2, 5** are almost identical with the steps that would be necessary in previous algorithms [8] with the exception that now the look up process is performed on a much smaller table with decreased access latency.

The quadrant determination of the initial edge vector specified by **step 1** can be implemented using the circuit presented in Figure 5. The 2-bit quadrant code assignment

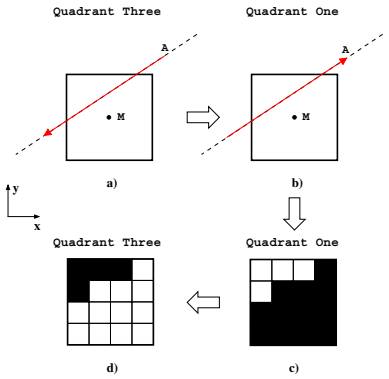

Figure 7. Q3 edge vector coverage mask generation.

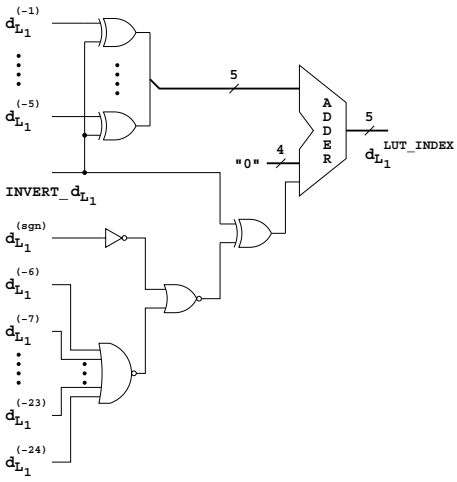

Figure 8. $\mathrm{d_{L_1}}$ selective sign complementation and truncate-to-zero circuit diagram.

is "00" for quadrant one (Q1), "01" for quadrant two (Q2), "10" for quadrant three (Q3), and "11" for quadrant four (Q4). An additional error signal is provided to flag degenerate edge vectors ($\Delta x = \Delta y = 0$) and in effect to disable the rasterization of such degenerate triangles. As this circuit is already employed by the point-sampling triangle rasterization datapath to impose tie-rules for pixel rasterization on triangle shared edges, it will not be considered in the followings as part of the antialiasing datapath.

The quadrant disambiguation (**step 3**) and the coverage mask transformation to the originary quadrant (**step 7**) are meaningful only if they are explained in synergy. The idea behind is to transform the arbitrary quadrant edge vector into an equivalent Q1 edge vector in order to use for coverage mask retrieval only a reduced coverage mask LUT for Q1 edge vectors. After the coverage mask is fetched from the LUT, inverse transformations have to be operated on the

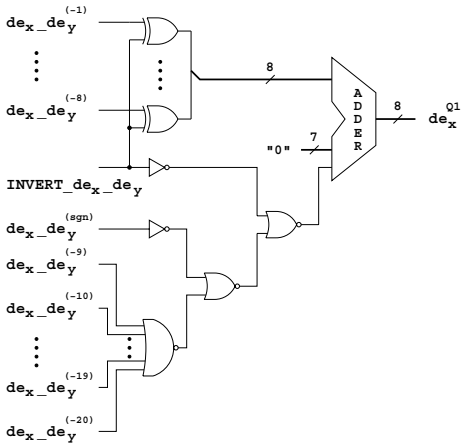

Figure 9. $\mathrm{de_x}$ or $\mathrm{de_y}$ selective sign complementation and truncate-to-zero circuit diagram.

coverage mask in order to obtain the correct coverage mask for the initial, arbitrary quadrant edge vector. The equivalent underlying geometrical transformations to the abovementioned formulas required to generate Q2 and Q3 coverage masks are depicted in Figure 6, respectively Figure 7. When the edge belongs to Q4 the operations required are fused computations Q4→Q2→Q1→Q2→Q4. The transformations for forward transition Q4→Q2 and backward transition Q2→Q4 are similar to Q3→Q1 and Q1→Q3 respectively. This forward/backward transformations ensure by construction that two adjacent triangles, both front-facing or both back-facing, always complement each other, and a total coverage of more than 4×4 subpixels is impossible meaning that the algorithm is water-tight. In the following, efficient circuits to implement **step 3** are presented. There are two problems to be tackled with when using two's complement number representation. The first one is the requirement for wide-operand addition to implement the sign complementation unary operator (for our required precision 26-bit addition for d_{L_1} and 22-bit addition for de_x). The second one and the only mean to warrant water-tightness, given the asymmetrical behavior of positive and negative numbers under truncation (required for **steps 4, 5**), is to employ a truncate-to-zero rounding scheme. This is accomplished by ignoring the least significant bits on the right side and adding the sign bit to the least significant bit of the remaining bits, however this only occurs if at least one of the ignored bits is nonzero. This involves a chain of two additions, one of them being expensive. To simplify things and reduce it to two narrow-operand additions, it can be shown that a sign complementation followed by a truncate-to-zero rounding is equivalent to a truncate-to-zero rounding first followed by the sign complementation of the resultant re-

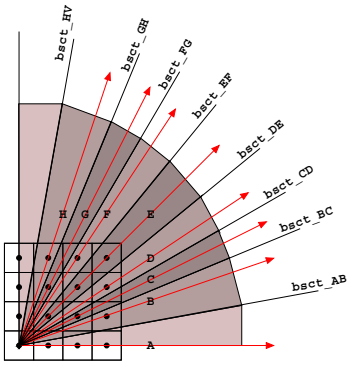

Figure 10. Edge vector class disambiguation employing bisectors.

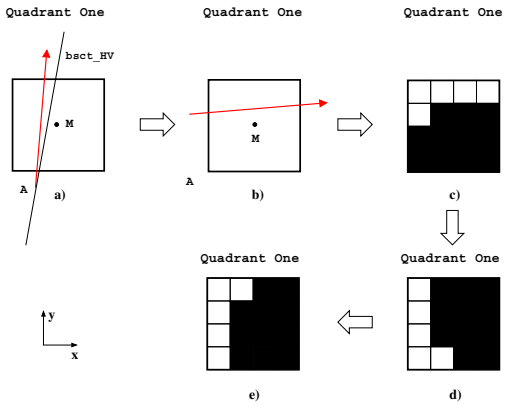

Figure 11. Coverage mask adjustment.

duced number of bits. Furthermore, it is possible to fuse these two additions in only one narrow-operand addition using the circuits presented in Figure 8 and Figure 9 making use of little additional logic and a signal that indicates if the sign complementation is required. This circuits eliminate the need for additional redundancy to be built in the coverage masks LUT as in [8], lowering the LUT foot-print further.

The role of the edge vector class disambiguation (**step 4**) is to map the parameters of the quadrant one edge vector resulted from the previous step (quadrant disambiguation — **step 3**) into parameters of the closest matching representative edge vector whose coverage mask is resident in LUT. The quadrant one edge vector that results after the quadrant disambiguation process (**step 2**) has to be classified in one of the eight quadrant one edge vector classes whose coverage masks are stored in LUT. Conceptually, the disambiguation process of the edge vector class is reduced to the problem of finding the boundaries between neighboring quadrant one edge vector classes with correspondence in the coverage masks LUT. This was solved by finding the $de_x^{Q1}(\alpha)$ value of the bisectors between two adjacent quadrant one edge vector classes with correspondence in LUT. Referring to Figure 10, it means that if the $de_x^{Q1}(\alpha)$ value of an incoming edge vector is, for example, between the $de_x^{Q1}(\alpha)$ values of the bisector `bsct_AB` and `bsct_BC`, then its $de_x^{Q1}(\alpha)$ value becomes that of the edge vector class `B`. Since only eight edge vector classes (`A`, `B`, `C`, `D`, `E`, `F`, `G`, `H`) are represented in the coverage mask LUT, it means that only 3 bits are needed to encode this value in the coverage mask LUT index. This 3-bit code is produced directly as a result of the edge vector class disambiguation with bisectors. In the coverage mask LUT being stored 256 coverage masks, 5 bits remain available (as in a previous implementation[8]) in the index to encode 32 L_1-norm distances $d_{L_1}(M)$ (coverage masks for 32 values of distances from the pixel center M to a particular edge slope can be stored). The rules for the edge vector class disambiguation with bisectors are presented in Table 1, column 1 and 2. It is needed to emphasize that the $de_x^{Q1}(\alpha)$ values associated with the bisectors represent constants to the algorithm which will be programmed in hardware and no computational effort is spent at rasterization time to compute them. The 3-bit code required to encode the disambiguated $de_x^{Q1}(\alpha)$ in the coverage mask LUT index is presented in Table 1, column 3. Referring to Figure 10, an exceptional case that have to be handled in a specific way appears for the disambiguation of any quadrant one edge vector class whose slope lies between `bsct_HV` and the vertical. Normally, it will have to be disambiguated to the vertical edge vector class but according to the assignment presented in Figure 4, this class belongs to the quadrant two. Instead, those exceptional edge vectors are disambiguated by wrapping around to the `A` edge vector class (last row in Table 1) and asserting a condition signal `wrap` (Table 1, column 4). The coverage mask is fetched from the coverage mask LUT, but before applying **step 7**, the correction described in **step 6** has to be performed if the condition signal is asserted. The equivalent underlying geometrical transformations for the coverage mask adjustment process are presented in Figure 11. The edge disambiguation rules presented in Table 1 can be implemented in two ways: sharing the gates for implementing the carry chains necessary for each required comparison or specifying the edge diambiguation rules in a logic table format with an entry for every possible $de_x^{Q1}(\alpha)$ value. Both approaches can be synthesized efficiently leading to a fast logic circuit, for example considering 8-bit disambiguation constants the resultant circuit complexity is slightly less than a 16-bit adder.

Step 8 is required in order for the coverage mask lookup scheme to work with triangles with edges oriented clockwise or counter-clockwise, as required for OpenGL or Microsoft's DirectX-Direct3D compliance. The coverage masks in the coverage mask LUT are computed only for

Table 1. Edge vector class disambiguation rules.

Range $\mathbf{de_x^{Q1}}(\alpha)$	Disambiguated $\mathbf{de_x^{Q1}}(\alpha)$	$\mathbf{de_x^{LUT\_index}}$ (binary)	wrap (binary)
$\left[0, de_x^{bsct\_AB}\right)$	de_x^A	000	0
$\left[de_x^{bsct\_AB}, de_x^{bsct\_BC}\right)$	de_x^B	001	0
$\left[de_x^{bsct\_BC}, de_x^{bsct\_CD}\right)$	de_x^C	010	0
$\left[de_x^{bsct\_CD}, de_x^{bsct\_DE}\right)$	de_x^D	011	0
$\left[de_x^{bsct\_DE}, de_x^{bsct\_EF}\right)$	de_x^E	100	0
$\left[de_x^{bsct\_EF}, de_x^{bsct\_FG}\right)$	de_x^F	101	0
$\left[de_x^{bsct\_FG}, de_x^{bsct\_GH}\right)$	de_x^G	110	0
$\left[de_x^{bsct\_GH}, de_x^{bsct\_HV}\right)$	de_x^H	111	0
$\left[de_x^{bsct\_HV}, +1\right)$	de_x^A	000	1

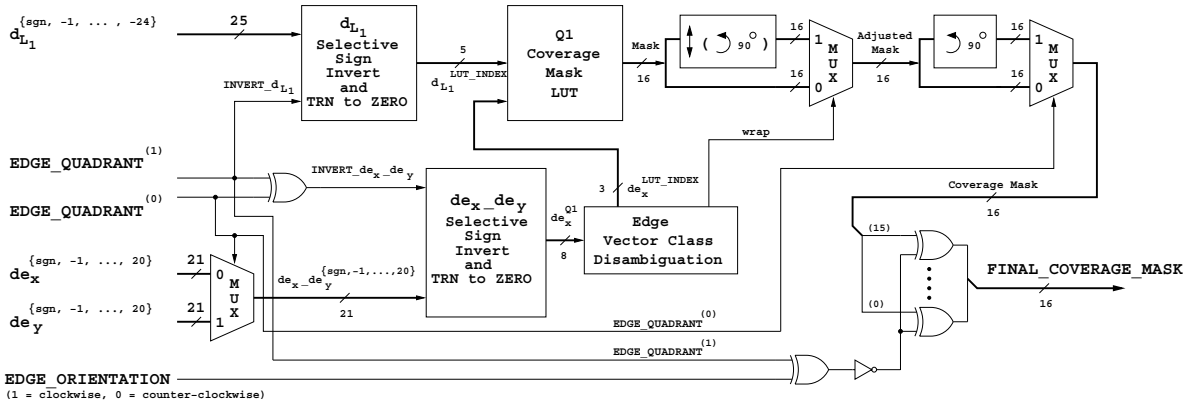

Figure 12. Coverage mask generation circuit diagram for one edge vector.

clockwise orientation of triangle's edge vectors. For triangles with edges oriented counter-clockwise the coverage mask obtained through the operations described so far has to be bitwise negated to deliver the final coverage mask. The orientation of the triangle's edges can be detected by computing in Equation (1) the sign of the edge function $E^{AB}(x_C, y_C)$, or equivalently, the normalized edge function $d_{L_1}^{AB}(x_C, y_C)$ (in any cyclic permutation of triangle's vertices A, B, C). Those computations are not specific to the antialiasing datapath, they being required mandatory for the triangle interpolation setup, i.e., $\delta z/\delta x$, $\delta z/\delta y$ etc.

A diagram of the entire coverage mask generation circuit for one edge vector is presented in Figure 12. The diagram corresponds to Figure 1(b). To summarize, the proposed algorithm leads to efficient hardware implementations having a lower structural cost and requiring only computationally inexpensive operations.

4. Hardware implementation and simulation results

A whole OpenGL-compliant 3D graphics rasterizer, including the proposed pixel coverage mask generation hardware algorithm (256 16-bit coverage masks), was modeled at RT-level in SystemC language [6]. Referring to the internal organization, the rasterizer adopts a tile-based rasterization approach. The tile size chosen for this particular implementation was set at 32×16 pixels which implies that all the internal buffers (color buffer, depth buffer, stencil buffer) composing the tile frame buffer have this size. The display size resolution was set at 320×240 pixels (QVGA), meaning that the display can be conceptually divided into 10×15 tiles. The rasterizer has only one pixel processing pipeline. The screen coordinates (X, Y) are represented on 9.4 bits (9 integer, 4 fractional), the color components (R,G,B,A) on 0.8 bits, the depth component (Z) on 0.24 bits, and the stencil component on 8.0 bits.

The "aapoly" OpenGL application from [9] was executed on the virtual graphics hardware rasterizer. The generated image is presented in Figure 13. The antialiasing image quality can be seen in the detailed regions featuring pixel center and primitive geometry overlaid markings. The results of the hardware synthesis using Synopsys tools in a commercial 0.18μm IC manufacturing technology of the coverage mask generation circuit for one edge vector are

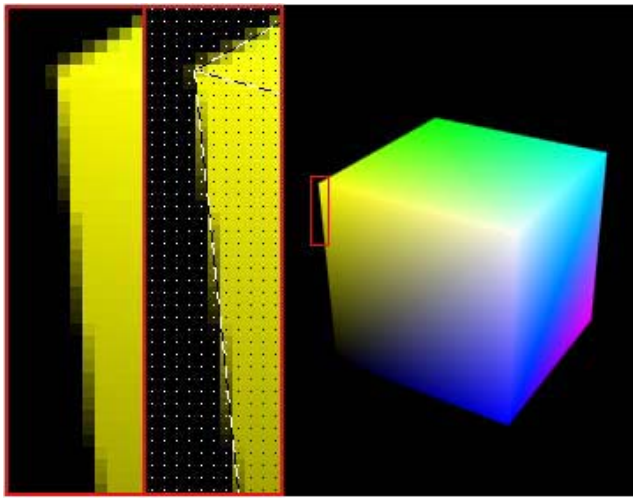

Figure 13. Antialiasing employing the proposed coverage mask generation hardware algorithm and implementation.

IC Technology		Std. Cell Library
UMC *Logic18-1.8V/3.3V-1P6M*		VST *e*Si-Route/11
ED Latency	**ED Std. Cell No.**	**ED Cell Area**
0.5ns	42	$833\mu\mathrm{m}^2$
Total Latency	**Total Std. Cell No.**	**Total Cell Area**
2.49ns	557	$12270\mu\mathrm{m}^2$

Table 2. Hardware synthesis results for the coverage mask generation circuit for one edge vector.

presented in Table 2. Results are also provided for the edge vector class disambiguation circuit with bisectors. The efficiency of the proposed implementation is difficult to be quantified in respect to past solutions that produced coverage masks using normalized edge functions given that they do not provide details about their hardware implementation. In an attempt to provide a fair comparison, we implemented the solution described in [8] and the results were an implementation with 8432 standard cells, an area of $270375\mu\mathrm{m}^2$, and a latency of 4.2ns. This result indicates that our implementation is much more efficient, in addition we managed to use the area recovered to implement the rest of the antialiasing hardware datapath specified by Equation (1).

5. Conclusions

An efficient low-cost, low-power hardware implementation of a novel run-time pixel coverage mask generation algorithm for embedded 3-D graphics antialiasing purposes has been presented. The algorithm is exploiting the quadrant symmetry property allowing the storage of only the coverage mask information for a few representative edges in one of the quadrants of the plane, the rest of the information being derived on the fly via computationally inexpensive operations. The algorithm was presented assuming 4×4 subpixel coverage masks and two's complement number arithmetic, however it has a higher degree of generality: it can be incorporated in any antialiasing scheme with pre-filtering that is based on algebraic representation of primitive's edges, it is independent of the underlying number representation, and it can be adapted to other coverage mask subpixel resolutions with the only prerequisite for the masks to be square. Assuming a 0.18μm IC manufacturing technology, hardware synthesis results of the coverage mask generation circuitry indicated that our algorithm requires an area of $12270\mu\mathrm{m}^2$ and has a latency of 2.49ns instead of $270375\mu\mathrm{m}^2$ and 4.2ns required by state of the art solutions [8].

References

[1] L. Carpenter. The A-Buffer, an Antialiased Hidden Surface Removal Method. *ACM SIGGRAPH '84 Conference Proceedings*, 18:103–108, 1984.

[2] D. Crisu, S. Cotofana, and S. Vassiliadis. A Proposal of a Tile-Based OpenGL-Compliant Rasterization Engine. Technical report, Computer Engineering Laboratory, Delft University of Technology, Deliverable no. (2002)–02, 2002.

[3] M. Deering and D. Naegle. The SAGE Graphics Architecture. In *Proceedings of ACM SIGGRAPH 2002*, pages 683–692, 2002.

[4] J. Foley, A. van Dam, S. Feiner, and J. Hughes. *Computer Graphics: Principles and Practice, Second Edition in C*. Addison-Wesley, 1996.

[5] H. Fuchs, J. Goldfeather, J. Hultquist, S. Spach, J. Austin, F. Brooks, J. Eyles, and J. Poulton. Fast Spheres, Shadows, Textures, Transparencies, and Image Enhancements in Pixel-Planes. *Computer Graphics (ACM SIGGRAPH '85 Conference Proceedings)*, 19(3):111–120, 1985.

[6] T. Grötker, S. Liao, G. Martin, and S. Swan. *System Design with SystemC*. Kluwer Academic Publishers, 2002.

[7] J. Pineda. A Parallel Algorithm for Polygon Rasterization. *Computer Graphics (ACM SIGGRAPH '88 Conference Proceedings)*, 22(4):17–20, 1988.

[8] A. Schilling. A New Simple and Effi cient Antialiasing with Subpixel Masks. *Computer Graphics (ACM SIGGRAPH '91 Conference Proceedings)*, 25(4):133–141, 1991.

[9] M. Woo, J. Neider, T. Davis, and D. Shreiner. *OpenGL Programming Guide, Third Edition, The Offi cial Guide to Learning OpenGL, Version 1.2*. Addison-Wesley, 1999.