# Design Tradeoffs for an Embedded OpenGL-Compliant Hardware Rasterizer

Dan Crisu, Sorin Cotofana, Stamatis Vassiliadis
Computer Engineering Laboratory, EEMCS
Delft University of Technology
Mekelweg 4, 2600 GA Delft, The Netherlands
Phone: +31 15 2783644 Fax: +31 15 2784898
E-mail: {dan|sorin|stamatis}@ce.et.tudelft.nl

Petri Liuha
NOKIA Research Center
Tampere, Finland
E-mail: petri.liuha@nokia.com

*Abstract*—**This paper addresses design trade-offs for low-power, low-cost embedded 3D graphics accelerators. More in particular it focuses on a low-cost reciprocation hardware algorithm suitable to be implemented in their datapath. The algorithm exploits the limitations of the human visual system that allows a reasonable amount of error to be introduced in the computation process without inducing noticeable artifacts in the final computer-generated image. In the example given in the paper, excerpted from the antialiasing datapath of an embedded QVGA graphics hardware accelerator, for a 14-bit operand, the reciprocal implementation requires an inexpensive operand prescaler, one 1k lookup table with 10-bit entries, and a 5-bit adder, for a maximum relative error of the result of only 1.5% over the entire range of the operand. Hardware synthesis in a typical $0.18\mu$m process technology has indicated that the hardware implementation requires only 1600 standard cells to achieve a latency of only one clock cycle for a clock frequency of 250MHz.**

*Keywords*—**graphics architecture; embedded systems; reciprocal approximation; system-on-chip; hardware/software co-simulation**

## I. Introduction

With the advent of increasing powerful mobile platforms for computing and communications, 3D computer graphics hardware acceleration seems to become the next generation integration target in these devices. Consequently, there is much interest in the embedded systems research community to provide low-power, cost-effective real-time 3-D computer graphics capabilities. This means that in contrast with mainstream graphics hardware accelerator implementations that provide datapath bit-exact arithmetic at a high expense regarding cost and power-consumption, embedded graphics hardware implementations have to exploit the limitations of the human visual system which allows a reasonable amount of error to be introduced in the computation process without introducing noticeable artifacts in the final computer-generated image. Following, in this paper arithmetic precision trade-offs are explored in the context of an OpenGL[1]-compliant rasterization engine hardware implementation we developed, meant to be integrated in next-generation ARM-based system-on-chip designs. The hardware model of the rasterization engine is specified in the SystemC language [2] and it is simulated using GRAAL (GRaphics AcceLerator) Simulator [3], our power-aware hardware/software co-simulation environment custom built for graphics hardware accelerator development for ARM-based system-on-chip designs. Using this tool and modifications of the standard OpenGL graphical pipeline embedded in the hardware model, graphical output is provided for the visualization of the potential impact tweaking the algorithms or the bit operand width precision may have on the resulted image quality, as well as a quantitative estimation of the errors gathered in the process. As a result, we can show that reducing the internal precision for certain operations and applying a number of computational approximations on commonly encountered and difficult arithmetic operators in graphics, like division, pertaining especially to triangle antialiasing, triangle setup, texture coordinates generation and texture mapping, benefic effects can be achieved such as area and power consumption reduction at the same performance level without compromising the image quality.

This paper presents a low-cost reciprocation hardware algorithm suitable to be implemented in the datapath of low-power, low-cost embedded 3D graphics accelerators. In the example given in the paper for a 14-bit operand, excerpted from the antialiasing datapath of an embedded QVGA graphics hardware accelerator, the reciprocal implementation requires an inexpensive operand prescaler, one 1k lookup table with 10-bit entries, and a 5-bit adder, for a maximum relative error of the result of only 1.5% over the entire range of the operand. Hardware synthesis in a typical $0.18\mu$m process technology has indicated that the hardware implementation requires only 1600 standard cells to achieve a latency of only one clock cycle for a clock frequency of 250MHz.

The rest of the paper is organized as follows. An

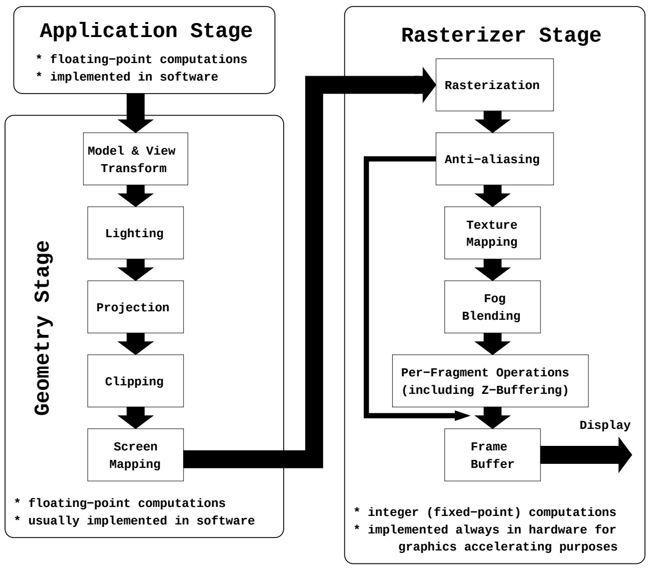
Application Stage
* floating-point computations
* implemented in software
Geometry Stage
Model & View Transform
Lighting
Projection
Clipping
Screen Mapping
* floating-point computations
* usually implemented in software
Rasterizer Stage
Rasterization
Anti-aliasing
Texture Mapping
Fog Blending
Per-Fragment Operations (including Z-Buffering)
Frame Buffer
Display
* integer (fixed-point) computations
* implemented always in hardware for graphics accelerating purposes

Fig. 1. A typical 3D graphics pipeline.

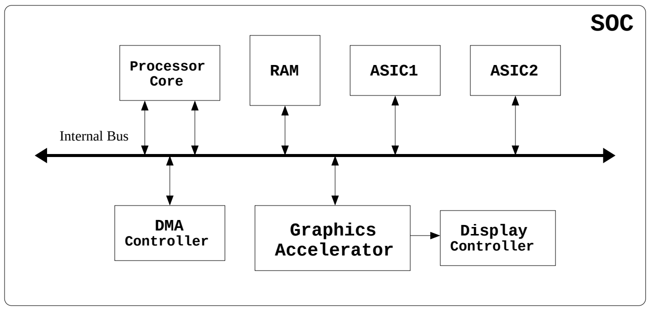

Fig. 2. SOC organization.

overview of an SOC platform for embedded graphics applications is presented in Section II. The role of division by reciprocation in certain sub-stages of the graphics pipeline (rasterization stage) is briefly discussed in Section III. The reciprocal hardware algorithm we propose for 3D graphics is presented in Section IV. The impact on the quality of the rendered images using the proposed algorithm and hardware synthesis results are presented in Section V. Finally, Section VI draws the conclusions.

## II. Embedded 3D graphics

A 3D graphics rendering system is organized conceptually as a number of stages chained in a pipelined fashion. The conceptual stages of the graphics pipeline are the *Application*, the *Geometry*, and the *Rasterizer Stage*. They are presented in Figure 1. An in-depth explanation of these stages is beyond the scope of the paper and the reader is referred for more details to any computer graphics textbook, e.g. [4].

Typically, a platform for embedded graphics applications looks like in Figure 2 and the mapping of the conceptual stages of the 3D graphics pipeline in the system are described in the sequel. The graphics software application is running on the host processor of the system. The software application corresponds to the conceptual Application Stage of the graphics pipeline. The software application is relying on a 3D graphics library (perceived in the sense of a software interface to the graphics hardware [5]) like OpenGL or Direct3D to have its graphic calls taken care further. This 3D graphics library executes usually the conceptual Geometry Stage on the host processor. The code that implements the Geometry Stage in the library can further make calls to the graphics hardware accelerator by means of a standardized, virtual interface, to ensure library portability. Between this virtual interface and the graphics hardware accelerator (on which the conceptual Rasterizer Stage is mapped) there is another piece of code executed on the host processor called a device driver. This device driver performs the function of a hardware abstraction layer and it translates the calls through the virtual interface in actual memory-mapped or programmable I/O instructions (seen from the host processor point of view) particular to the graphics hardware accelerator's input and output register port mapping in the system address space. Finally, the Rasterizer Stage is executed in hardware on the graphics hardware accelerator (due to the computational explosion at this level) performing the following operations. Given the primitives (usually triangles) received from the host processor with transformed and projected vertices, colors, and texture coordinates computed for this vertices, the goal of the Rasterizer Stage inside the graphics accelerator is to assign correct colors to the pixels that will be stored in a memory called the *frame buffer*, which is read periodically by the display controller to form the image on the screen. This process is called *rasterization* or *scan conversion*. During rasterization, the information needed for the screen pixels covered by the primitive is interpolated from the data (depth, colors and texture coordinates) associated with its projected vertices on the screen. In this way series of frame buffer addresses and values called *fragments* are produced for each rasterized primitive. Each fragment so produced is fed to the next functional stage depicted in Figure 1 that performs operations on individual fragments before they finally alter the frame buffer. These operations include color alteration based on the textures assigned per primitive and texture coordinates, fog blending, conditional updates into the frame buffer based on incoming and previously stored depth values $z$ in the *depth buffer* or $z$-*buffer*, blending of incoming fragment colors with stored colors, as well as masking and other logical operations on fragment values. Due to the sampling process involved by rasterization, a

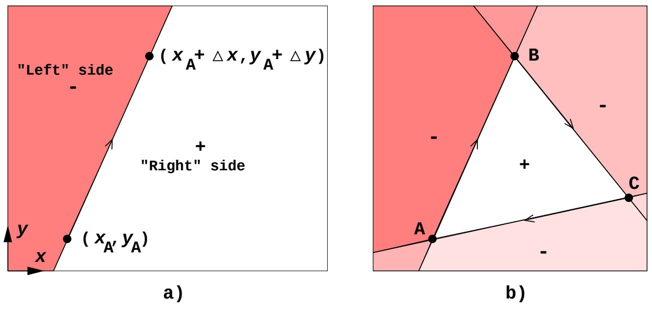

Fig. 3. Triangle representation using edge functions: a) The edge is defined by a vector starting at the point $A$ with the slope $\Delta y/\Delta x$, b) The interior of the triangle is formed by the union of the right sides of the oriented edges $AB$, $BC$, and $CA$.

chain of filtering operations followed by resampling may be necessary on fragment values to alleviate the inherent aliasing phenomenon (e.g., the staircasing effect of lines drawn on a raster screen). Finally, the fragments that will survive in the frame buffer, after all of the primitives have been processed, will produce the final image. All the operations required by the Rasterizer Stage usually involves only integer (fixed-point) arithmetic.

## III. Reciprocal operations in rasterization

The rasterization pipeline in 3D computer graphics requires several reciprocal computations along the way. Two examples will be presented in the following. The enumeration is by no means exhaustive, for more in-depth information the reader being referred to [6].

### A. Triangle antialiasing with prefiltering

One method of triangle rasterization is based on the algebraic representation of triangle's edges. Given that a triangle edge can be represented as a vector between its respective vertices (one being the source and the other the sink), it can be detected if the current rasterization position $(x_M, y_M)$ in the screen space lies in one of the half-planes delimited by the edge vector or exactly on the edge vector. The detection process involves the computation of the cross-product of the edge vector with the vector formed by the edge vector's source and the current rasterization position $(x_M, y_M)$. This cross-product can be expressed for any arbitrary pixel position $(x, y)$ on the screen as an analytical function, called the edge function, in the unknowns $(x, y)$. When the edge function is evaluated at the current rasterization position $(x_M, y_M)$, if the result of the evaluation is zero than the position $(x_M, y_M)$ lies exactly on the edge, or, depending on the sign of the evaluation's result, the position $(x_M, y_M)$ lies in one of the half-planes delimited by the edge vector. The half-planes delimited by the edge and their connection with the edge function value evaluated for points lying in the plane are presented in Figure 3(a). Considering a triangle described by its oriented edge vectors, a position belongs to the interior of a triangle if all its edge functions computed for that position have the same sign. Therefore, the values for the edge functions are used as a *stencil* (depicted in Figure 3(b)) in rasterization that allows a pixel to be modified only if it is interior to the triangle. Additionally, if the edge function is properly normalized, its evaluation yields the distance from the edge vector to the pixel position $(x, y)$ that can be used in determining the coverage of the triangle's edge over the pixel position $(x, y)$. This means that the *normalized* edge function can be employed for both rasterization and antialiasing. Such a scheme was presented in the Exact Area Sampling Algorithm (EASA) proposed by Schilling [7] based on the Pineda's effective edge function formulation [8]. The normalized edge function formulation of EASA is as follows:

$$\begin{aligned} d_{L_1}(M) &= \frac{E(x_M, y_M)}{|\Delta x| + |\Delta y|} \\ &= (x_M - x_A) \cdot \frac{\Delta y}{|\Delta x| + |\Delta y|} - (y_M - y_A) \cdot \frac{\Delta x}{|\Delta x| + |\Delta y|} \\ &= (x_M - x_A) \cdot de_x(\alpha) - (y_M - y_A) \cdot de_y(\alpha) \end{aligned} \tag{1}$$

It can be seen in Equation (1) that a reciprocal is required by $de_x(\alpha)$ and $de_y(\alpha)$ parameter computation.

### B. Triangle setup

To rasterize a triangle, the exact values for the edge functions, $z$, colors, and texture coordinates are computed for a conveniently chosen pixel $(x, y)$ on the screen and also interpolation steps (gradients) along the $x$ and $y$ axes are found for them. This stage is called the *triangle setup stage*. Then, the values for the adjacent pixels can be computed by simple linear interpolators that require only one addition per component per iteration. In the triangle setup stage reciprocal computations are required. The expressions for the gradient setup used in the depth ($z$ value) linear interpolation during the rasterization of the triangle with the vertices $A$, $B$, $C$ are:

$$\begin{aligned} \frac{\delta z}{\delta x} &= \frac{(\Delta y_{CA} \cdot \Delta z_{AB} - \Delta z_{CA} \cdot \Delta y_{AB})}{E_{AB}(x_C, y_C)} \\ \frac{\delta z}{\delta y} &= \frac{(\Delta z_{CA} \cdot \Delta x_{AB} - \Delta x_{CA} \cdot \Delta z_{AB})}{E_{AB}(x_C, y_C)} \end{aligned} \tag{2}$$

where $E_{AB}(x_C, y_C)$ represents the expression $E(x_C, y_C)$ of Equation (1) for the oriented edge $AB$.

The reciprocal of the $E_{AB}(x_C, y_C)$ expression is also required by the texture coordinates setup [6].

## IV. Reciprocal algorithm for 3D graphics

Division, being the most complex of the four basic arithmetic operations and the most difficult to speed up, has received a great deal of attention in the literature [9][10][11][12]. The more conventional approach uses add/subtract and shift operations with the operation count linearly proportional to the word size $n$. The second approach relies on multiplication and the number of steps performed is logarithmically proportional to $n$, but each individual step is more complex.

In this section we will focus on reciprocal computation (because in the rasterization pipeline many divisions have a common denominator) thus transforming the division in a multiplication of the numerator with the reciprocal of the denominator. Even so, this division method is more expensive and slower than a multiplication. Therefore, instead of providing datapath bit-exact arithmetic at a high expense regarding cost, latency, and power-consumption, we will try to reduce the cost of the reciprocal by exploiting the limitations of the human visual system which allows a reasonable amount of error to be introduced in the computation process without introducing noticeable artifacts in the final computer-generated image.

Defining the relative error $\epsilon_{rel}(1/X)$ of the reciprocal computation as:

$$\epsilon_{rel}\left(\frac{1}{X}\right) = \frac{\left(\frac{1}{X}\right)_{approx} - \frac{1}{X}}{\frac{1}{X}} \tag{3}$$

where $(1/X)_{approx}$ represents the reciprocal computed in hardware and $(1/X)$ represents the true value of the reciprocal of a given value $X$, our experiments on a QVGA display have suggested that a relative error of maximum 5% does not introduce visible artifacts in the generated image.

In the followings, to illustrate the different design tradeoffs in the design of the reciprocal we will work on the example given in Subsection III-A. The screen coordinates $x$ and $y$ for a QVGA display (with a resolution of $320 \times 240$) will be represented as unsigned fixed-point numbers in the format $9.4$. The fractionary part of the coordinates is necessary to eliminate drop-outs and overlaps in the rasterized image [13]. This means that the quantity $|\Delta x| + |\Delta y|$ will be represented as an unsigned fixed-point number in format $10.4$. Moreover, triangles will be culled in the software driver if $|\Delta x| + |\Delta y| < 2^{-4}$ imposing that the reciprocal should be less than or equal to $2^4$. Also, $|\Delta x| + |\Delta y| \leq (2^{10} - 2^{-4})$ establishing the minimum value of the reciprocal to approximative $2^{-10}$.

To summarize, the objective is to compute the reciprocal of an unsigned non-zero fixed-point number $X$ represented in a $10.4$ format with a maximum relative error of only 5%. In the following, three methods involving table lookup are examined.

| Index($X$) | Reciprocal $\left(\frac{1}{X}\right)$ | Entry used |
|---|---|---|
| 0000000000.0000 | 11111.1111111111 | No |
| 0000000000.0001 | 10000.0000000000 | Yes |
| 0000000000.0010 | 01000.0000000000 | Yes |
| 0000000000.0011 | 00101.0101010101 | Yes |
| ... | ... | ... |
| 1111111111.1110 | 00000.0000000001 | Yes |
| 1111111111.1111 | 00000.0000000001 | Yes |

TABLE I
LOOKUP TABLE CONTENT FOR METHOD 1.

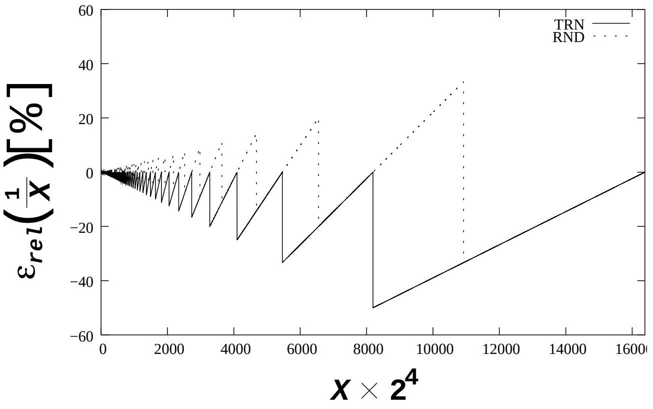

Fig. 4. Reciprocal relative error for Method 1 using reciprocal truncation (TRN) or rounding (RND) in the lookup table.

### A. *Method* 1

The first method presented implements the reciprocal using direct table lookup. Employing the results of the analysis performed in the beginning of the section, the reciprocal can be represented in fixed-point format with $5$ bits for the integer part and $10$ or more bits for the fractionary part. Choosing a minimal representation for the fractionary part with $10$ bits, the lookup table size required is 16k ($x$ is represented with $14$ bits) entries of 15-bit each. The content of the lookup table is presented in Table I. The relative error of the reciprocal computation using this method is depicted in Figure 4. The errors are significant, being larger than 5%.

### B. *Method* 2

The reciprocal relative error can be drastically reduced if, instead of storing the reciprocal in fixed-point format, it is stored in floating-point with a $1.6$ fixed-point format for the mantissa and $5$ bits for the exponent in two's complement. All the entries should be normalized in order for their most significant bit in mantissa to be $1$ thus eliminating one bit per entry. The content of the lookup table

| Index($X$) | Reciprocal $\left(\frac{1}{X}\right)$ Mantissa | Exponent | Entry used |
|---|---|---|---|
| 0000000000.0000 | (1).111111 | $-1$ | No |
| 0000000000.0001 | (1).000000 | $+4$ | Yes |
| 0000000000.0010 | (1).000000 | $+3$ | Yes |
| 0000000000.0011 | (1).010101 | $+2$ | Yes |
| ... | ... | ... | ... |
| 1111111111.1110 | 1.000000 | $-10$ | Yes |
| 1111111111.1111 | 1.000000 | $-10$ | Yes |

TABLE II
LOOKUP TABLE CONTENT FOR METHOD 2.

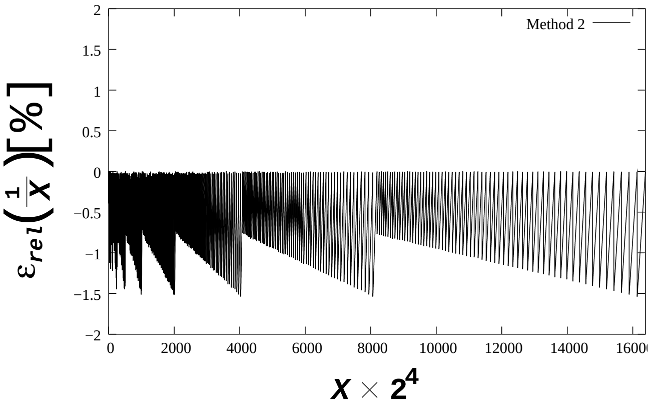

Fig. 5. Reciprocal relative error for Method 2 using reciprocal floating-point representation truncation in the lookup table.

is presented in Table II. The relative error of the reciprocal computation using this method is depicted in Figure 5. By using a floating-point representation in the lookup table, Method 2 is significantly more accurate than Method 1, the errors being within 1.5%. However, the lookup table is large requiring 16k 11-bit entries and its size reduction will be addressed in the next subsection.

### C. *Method* 3

Due to its nonlinearity, the reciprocal computation can be implemented using only a lookup table of 1k entries and a little amount of additional logic by employing the scheme depicted in Figure 6. The lookup table for reciprocal is presented in Table III and stores values only for the reciprocal of positive integers that can be represented on 10 bits. Now all the exponents are non-positive numbers and to save an extra bit per entry they can be inverted and stored as unsigned numbers. To loose as little precision as possible in the reciprocal computation the original 14-bit denominator has to be prescaled by left shifts up to 4 bit positions whenever the most 4 significant bits contain leading zeros. After shifting or even when no shifting is

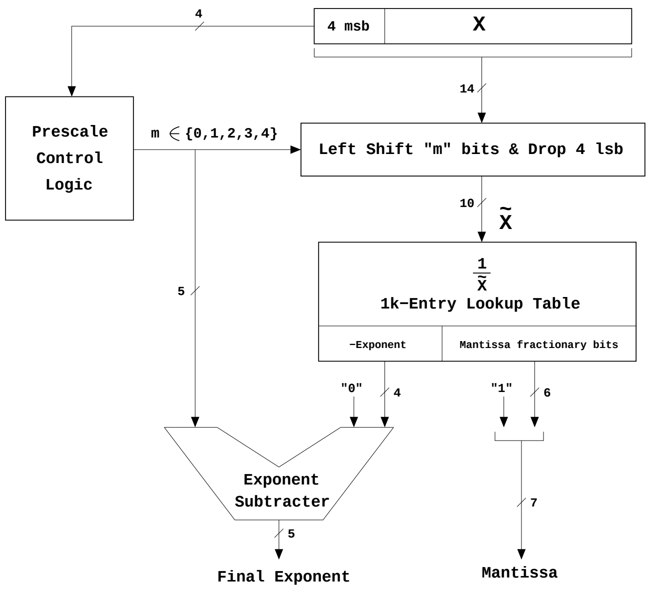

Fig. 6. Block diagram of the reciprocal algorithm employing a 1k-entry lookup table.

performed, the least four significant bits of the denominator are thrown away and the the surviving 10 bits in the result are used as an index in the lookup table to fetch the reciprocal. This is the role of the prescaling control logic and shifter depicted in Figure 6. The number of shifts performed are recorded and sent to a small 5-bit two's complement subtracter, also sketched in Figure 6, which compensates the inverted exponent fetched from the lookup table forming the true exponent of the reciprocal. Using this algorithm, the reciprocal of small numbers is performed with the accuracy given by the lookup table and for large numbers some or all of their 4 bits of fractionary part are ignored during computation leading to an insignificant additional error. The relative error of the reciprocal computation using this method over the entire range of possible 14-bit values of the denominator is depicted in Figure 7. Comparing Figures 7 and 5, it can be seen that the relative error for Method 3 is almost the same with the relative error given by Method 2 with the notable difference that Method 3 uses a 1k 10-bit entry lookup table instead of a 16k 11-bit entry lookup table. The overhead in cost introduced by the additional logic required by Method 3 is totally insignificant when compared with the size of the 1k-entry lookup table. Thus, Method 3 offers a low-cost hardware algorithm for reciprocation suitable to embedded 3D graphics.

## V. RESULTS

To assess the effectiveness of Method 3 presented in Subsection IV-C compared to Method 1 described in Subsection IV-A for reciprocal computation in 3D graphics,

| Index$\left(\tilde{X}\right)$ | Reciprocal $\left(\frac{1}{X}\right)$ | | Entry used |
|---|---|---|---|
| | Mantissa | $-$Exponent | |
| 0000000000 | (1).111111 | +15 | No |
| 0000000001 | (1).000000 | 0 | Yes |
| 0000000010 | (1).000000 | +1 | Yes |
| 0000000011 | (1).010101 | +2 | Yes |
| ... | ... | ... | ... |
| 1111111110 | (1).000000 | +10 | Yes |
| 1111111111 | (1).000000 | +10 | Yes |

TABLE III
LOOKUP TABLE CONTENT FOR METHOD 3.

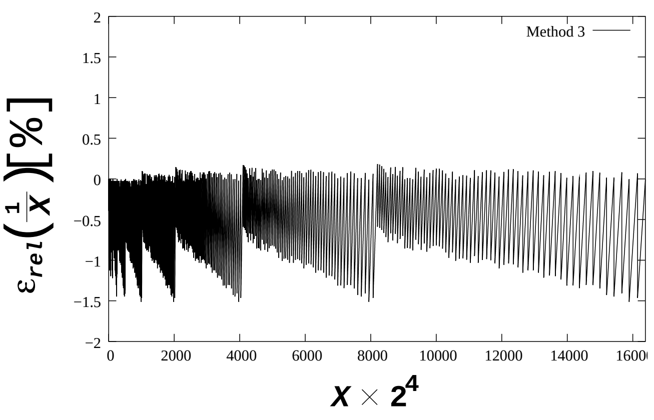

Fig. 7. Reciprocal relative error for Method 3.

we have modeled in SystemC RTL the reciprocation hardware. The hardware model was employed in the datapath of our OpenGL 1.2 compliant 3D graphics hardware accelerator SystemC model for an ARM based SOC platform. The graphics hardware accelerator supports the following 3D OpenGL rasterization functionality:

• Triangle rasterization: flat- and Gouraud-shaded with/without antialiasing with all the options controlling rasterization;
• Texturing with only RGBA8 internal texture format, texture fetching on demand;
• Per-fragment operations: scissor test, alpha test, stencil and depth buffer test, blending, logical operation;
• Whole frame buffer operations: fine control of buffer updates, clearing the buffers;
• State management: all the state management for previously mentioned functionality respecting all the invariance rules imposed by the OpenGL 1.2 specification;

The other primitives supported by OpenGL (points, lines, polygons with more than three vertices) are processed by the software driver and presented to the graphics hardware accelerator as a combination of triangles.

| IC Technology | | Std. Cell Library |
|---|---|---|
| UMC *Logic18-1.8V/3.3V-1P6M* | | VST *e*Si-Route/11 |
| Latency at Clk. Freq. | Std. Cell No. | Total Cell Area |
| 1 cycle at 250MHz | 1615 | $65000\mu\text{m}^2$ |

TABLE IV
METHOD 3 RECIPROCAL HARDWARE SYNTHESIS RESULTS.

Referring to the internal organization, the graphics accelerator adopts a tile-based rasterization approach. The tile size chosen for this particular implementation was set at $32 \times 16$ 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 $320 \times 240$ pixels (a quarter VGA), meaning that the display can be conceptually divided into $10 \times 15$ tiles. The graphics accelerator has only one pixel processing pipeline. The fixed-point formats utilized at the interface with the internal datapath are all unsigned. The screen coordinates (X, Y) are represented on $9.4$ bits, 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 [5] was executed on our virtual SOC platform. The resultant image is presented in Figure 8. It can be seen that the reciprocal employing Method 1 leads to noticeable artifacts in the image, whereas the reciprocal using Method 3 does not introduce artifacts in the image.

The results of the hardware synthesis on the reciprocal SystemC RTL model using the algorithm outlined in Method 3 are presented in Table IV.

## VI. CONCLUSIONS

This paper presented a low-cost reciprocation hardware algorithm suitable to be implemented in the datapath of low-power, low-cost embedded 3D graphics accelerators. The algorithm exploits the limitations of the human visual system that allows a resonable amount of error to be introduced in the computation process without inducing noticeable artifacts in the final computer-generated image. In the example given in the paper, excerpted from the antialiasing datapath of an embedded QVGA graphics hardware accelerator, for a $14$-bit operand, the reciprocal implementation requires an inexpensive operand prescaler, one 1k lookup table with 10-bit entries, and a 5-bit adder, for a maximum relative error of the result of only $1.5\%$ over the entire range of the operand. Hardware synthesis in a typical $0.18\mu$m process technology has indicated that the hardware implementation requires only 1600 standard cells to achieve a latency of only one clock cycle for a clock fre-

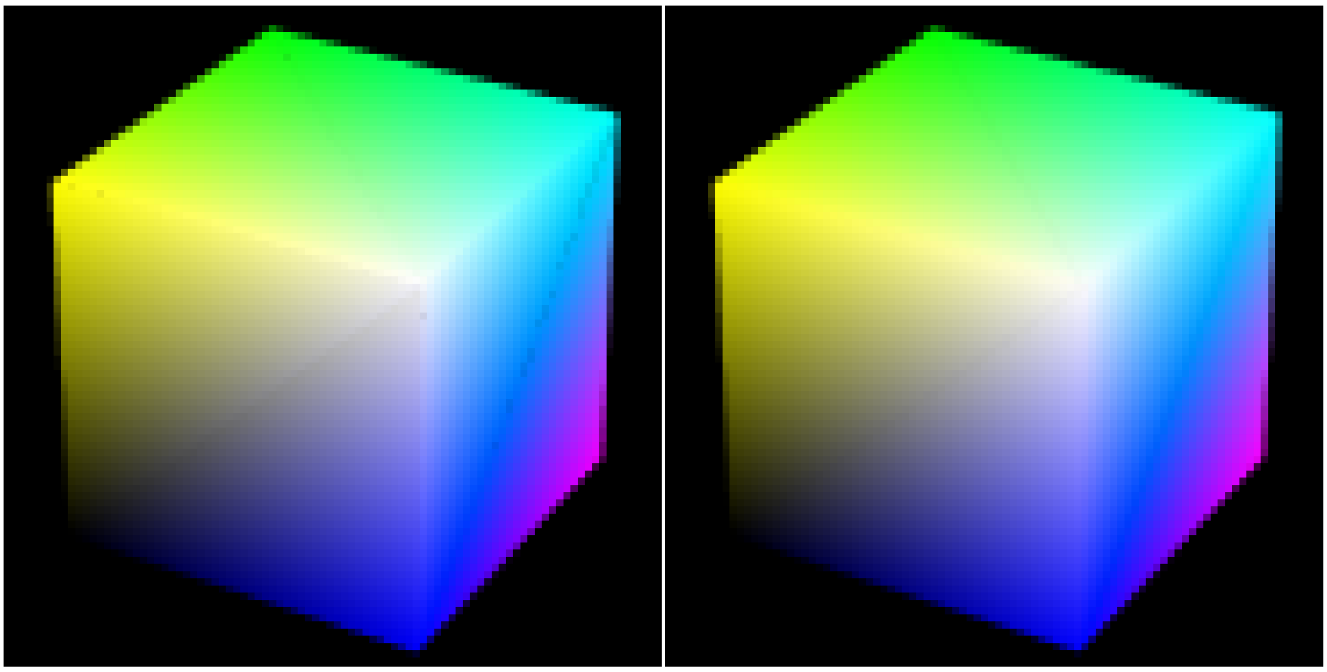

Fig. 8. A blowup on the image generated by “aapoly” OpenGL application — Left image: reciprocal computation using Method 1 produces noticeable artifacts; Right image: no artifacts produced when employing reciprocal computation according to Method 3.

quency of 250MHz.

## References

[1] M. Segal and K. Akeley. *The OpenGL Graphics System: A Specification (Version 1.2.1)*. Silicon Graphics, Inc., 1999.

[2] The Open SystemC Initiative (OSCI), URL: http://www.systemc.org.

[3] D. Crisu, S.D. Cotofana, and S. Vassiliadis. A Hardware/Software Co-Simulation Environment for Graphics Accelerator Development in ARM-Based SOCs. In *Proceedings of 13th Annual Workshop on Circuits, Systems and Signal Processing, ProRISC 2002*, November 2002.

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

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

[6] D. Crisu, S. Cotofana, and S. Vassiliadis. A Proposal of a Tile-Based OpenGL compliant Rasterization Engine. Technical Report (2002-02), Computer Engineering Laboratory, EEMCS, Delft University of Technology, June 2002.

[7] A. Schilling. A New Simple and Effi cient Antialiasing with Subpixel Masks. In *Computer Graphics (ACM SIGGRAPH ’91 Conference Proceedings)*, volume 25(4), pages 133–141, 1991.

[8] J. Pineda. A Parallel Algorithm for Polygon Rasterization. In *Computer Graphics (ACM SIGGRAPH ’88 Conference Proceedings)*, volume 22(4), pages 17–20, 1988.

[9] M. D. Ercegovac and T. Lang. *Division and Square Root:Digit-Recurrence Algorithms and Implementations*. Kluwer Academic Publishers, 1994.

[10] B. Parhami. *Computer Arithmetic: Algorithms and Hardware Designs*. Oxford University Press, 2000.

[11] M. J. Flynn and S. F. Oberman. *Advanced Computer Arithmetic Design*. John Wiley & Sons, 2001.

[12] I. Koren. *Computer Arithmetic Algorithms*. A K Peters, 2002.

[13] O. Lathrop, D. Kirk, and D. Voorhies. Accurate Rendering by Subpixel Addressing. *IEEE Computer Graphics and Applications*, 10(5):45–53, September/October 1990.