Heuristic Algorithms for Primitive Traversal Acceleration in Tile-Based Rasterization

Linjia Huang, Dan Crisu, Sorin Cotofana
Computer Engineering Laboratory
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
Mekelweg 4, 2600 GA Delft, The Netherlands
Phone: +31 15 2783644 Fax: +31 15 2784898
E-mail: {lhuang|dan|sorin}@ce.et.tudelft.nl

Abstract— **This paper addresses a series of hardware algorithms to reduce the computational overhead to locate the first rasterization tile position inside the primitive to be rasterized when the tile-based rasterization adopts the classical primitive traversal algorithm. These algorithms can be applied sequentially in a simple-to-complex order for searching a suitable starting tile rasterization position inside the primitive as follows: check if any of the vertices is in the tile, check if the triangle center of gravity (COG) is in the tile, recursive tile quadrant division based on COG attractors, and partial tile boundary scan. The algorithms were modeled in SystemC at the RT-level and integrated in a full-fledged OpenGL-compliant hardware rasterizer SystemC model. Simulation results on a benchmark suite consisting of 30 OpenGL applications have indicated that the throughput penalty is reduced to about 7% at the expense of about 10% increase in the hardware area when the entire OpenGL-compliant hardware rasterizer is synthesized in a commercial 0.18μm process technology.**

Keywords— **3D graphics architectures; tile-based rasterization; embedded systems; digital logic design**

I. Introduction

In recent years, with the increasing demand for graphics performance on mobile electronics, such as mobile phone and personal digital assistance (PDA), 2D/3D graphics computer hardware acceleration has become the next generation integration target for these devices. However due to the fact that the available hardware accelerators in market are mainly supporting the 2D graphics operations in hardware, such platforms are not able to provide the performance required by many of the state of the art 3D graphics applications. Thus the lack of explicit hardware support for 3D graphics operations make mobile platforms based on such solutions not an effective approach when 3D graphics applications are considered. Moreover, when such solutions are considered, the explosive increase in computations associated with the 3D rasterization process induces an almost unacceptable increase in the power consumption. This high power consumption is brought by the huge amount of arithmetic computations involved in the rasterization process and has been a notorious problem for designing 2D/3D graphics chips for portable devices. To deal with such a problem the designer has to give special attention to low power design aspects at various abstraction levels starting with the general architecture level and going down to layout and fabrication technology. The GRAAL project (GRAphics AcceLerator) [1] was initiated in order to provide such a power effective solution for 3D graphics acceleration. GRAAL is an OpenGL compliant tile-based rasterization engine, which employs versatile hardware-aware techniques. According to the simulation results produced by the GRAAL hardware/software co-simulation environment [2], the GRAAL engine can correctly render 3D images produced by an OpenGL application on a screen with good expected performance and power consumption. However, as previous investigations indicate [1], the utilization of a traditional primitive rasterization algorithm [3] in conjunction with the tile based architecture induces a computational overhead of 40%–300% associated to the location of an initial rasterization point inside the tile.

This paper presents a series of hardware algorithms to reduce the computational overhead associated with the location of a first rasterization position in the tile, also called hit point in the following of this paper. When utilized in a simple-to-complex sequence, they are Triangle Vertex Check, Triangle Gravity Center Check, Quadrant Search, and Backup Strategy. The primary simulation results on a set of OpenGL applications have indicated that the overhead is reduced to 3%–155% with an average of 7%. Hardware synthesis in a typical 0.18μm process technology has suggested a 10% increase in hardware area to the GRAAL engine.
The rest of this paper is organized as follows. After in-

troducing the principles of tile-based rasterization and the problem of the current primitive traversal algorithm in Section II, we present the heuristic algorithms used to solve the problem in Section III. The experimental simulation results are presented in Section IV, and conclusions are given in Section V.

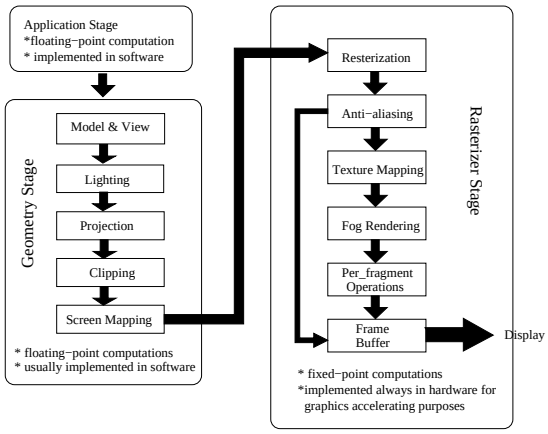


Fig. 1. A typical 3D graphics pipeline.

II. Background

A typical 3D graphics system conceptually consists of a number of stages that are chained in a pipeline style. The stages of this graphics pipeline are Application Stage, Geometry Stage, and Rasterizer Stage, see Figure 1. An in-depth explanation of these stages is beyond the scope of this paper and reader can be referred for more details to a computer graphics book, e.g. [4]. Typically, the application stage transforms objects into primitive models, such as point, line and triangle. The geometry stage is executed based on a 3D graphics library, such as OpenGL, on a host processor. The main task of the geometry stage is to generate transformed and projected vertices coordinates, colors, and texture coordinates, called geometrical data. Given these geometrical data, the goal of the rasterizer stage in a graphics accelerator is to assign correct colors to the pixels in order to render an image correctly. Generally, the application stage and the geometry stage are implemented in software, but the rasterizer stage is executed in hardware on the graphics hardware accelerator, due to the computational explosion at this level. In the rasterizer stage, fragment attribute values must be generated for each pixel position within the region of a primitive object, often a triangle. A fragment contains all the information required to render the surface at the pixel position, such as color, Z depth, texture coordinates, etc. The half plane edge function, presented in Equation 1, has been utilized in the GRAAL rasterization engine to produce the correct stencil of the primitive [3]:

$$E(x+\delta x, y+\delta y) = E(x,y) + \delta x \cdot \Delta y - \delta y \cdot \Delta x \quad (1)$$

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. Moreover, when a position's edge function results in zero, it means that the pixel lies on the edge.

Due to the usage of tiling architecture [4] by the GRAAL rasterization engine, the computation of the attribute values of the valid fragments (locations inside the primitive) has to be performed within the region of the current processed tile. Any algorithm that guarantees to cover all pixels of the tile that have a relationship with a triangle can be employed to traverse the triangle. As a preliminary solution, an exhaustive triangle traversal algorithm has been utilized to cover all the pixels in the triangle by visiting all the pixels in a tile from the left-bottom corner to the right-top corner, illustrated in Figure 2. It is easy to figure out that this algorithm is an inefficient way to traverse the triangle because it computes the three edge functions for pixels that may not belong to the triangle and therefore it has a lot of computational overhead. To speed up the algorithm, a second solution was to track only positions inside the triangle by examining the edge function signs once a hit position was found. However, there is remaining overhead in trying to detect the hit position, for example, see the path $P_{init} \rightarrow Q_{hit}$. This computational overhead accounts to 40%–300% of the primitive rasterization time. We will introduce in Section III various heuristic algorithms meant to speed up the location of the hit point.

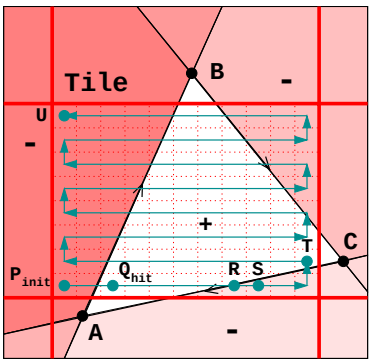


Fig. 2. Exhaustive tile traversal.

III. Hit Point Location Algorithms

To speed up the detection of a hit point in triangle traversal, several heuristic algorithms are presented in this section. When summarized in simple-to-complex order, these algorithms are Triangle Vertex Check, Triangle Gravity

Center Check, Quadrant Search Heuristics, and Backup Strategy. The idea is, considering that a triangle may assume any position in relation to a tile boundary, to try hit point candidates whose coordinates are directly available or can be derived with simple computations, before falling back to the straightforward solution depicted in Figure 2.

A. Triangle Vertex Check

Our investigations indicate that an average of 80% of triangles have at least one vertex inside the considered tile. Subsequently, triangle vertices can be taken as a promising candidate for a potential nearby hit point. The strategy to detect whether a triangle has at least one vertex interior to the current processing tile is to compare each of the triangle's vertex with the coordinates of the tile borders. The comparison is performed as follows:

IF $((X_{LEFT} \leq x_{vertex} \leq X_{RIGHT})$ AND
$(Y_{TOP} \leq y_{vertex} \leq Y_{BOTTOM}))$ THEN
$x_{hit} = x_{vertex}$;
$y_{hit} = y_{vertex}$;

where, the couple (x_{vertex}, y_{vertex}) is one of triangle vertex's coordinates, and the $X_{LEFT}, X_{RIGHT}, Y_{TOP}$, Y_{BOTTOM} are the coordinates of the tile borders. Finally, the (x_{hit}, y_{hit}) is the position of a hit point. The above comparisons are substituted in hardware by verifing that the tile index portions of the triangle vertex coordinates are identical with the tile indices.

B. Triangle Gravity Center Check

Given that there is still a chance that a hit point cannot be found among the triangle vertices, we have to consider other candidates too. Keeping in mind that the candidates should be obtained without too many computations, we have studied some other points of a triangle, such as gravity center, the intersection of triangle bisectors, triangle circumcenter, and orthocenter. From the study, we notice that all of centers, except of triangle gravity center, require complicated computations.
The triangle gravity center with screen coordinates (x_{gc}, y_{gc}) is computed by the following formulas:

$$x_{gc} = \frac{x_A + x_B + x_C}{3}, \tag{2}$$

$$y_{gc} = \frac{y_A + y_B + y_C}{3}. \tag{3}$$

Only 4 additions and 2 multiplications with the constant $1/3$ are required. This doesn't introduce much overhead of computation. Moreover, the triangle gravity center possesses another favorable property that it is interior to the triangle. Thus even if the gravity center fails to be a hit point, it can act as an attractor, guiding other approaches to reach the interior of a triangle.
The investigation on the relationship between a triangle gravity center and the current processing tile suggests that more than 50% of gravity center locates in the considered tile. Therefore, the triangle gravity center is another candidate for a hit point.
Same comparing strategy has been employed to check if a triangle's gravity center is interior to the processing tile. The comparison is performed as follows:

IF $((X_{LEFT} \leq x_{gc} \leq X_{RIGHT})$ AND
$(Y_{TOP} \leq y_{gc} \leq Y_{BOTTOM}))$ THEN
$x_{hit} = x_{vertex}$;
$y_{hit} = y_{vertex}$;

C. Quadrant Search Heuristics

When the triangle vertices and triangle gravity center fail to indicate a hit point, other approaches must be employed to continue the search of hit point. The *Quadrant Search Heuristics* is proposed for this purpose. The principle of the quadrant search heuristics can be described as follows:

To reduce the searched area partition the tile in quadrants, select the quadrant closer to the triangle gravity center and consider the center of such a quadrant a potential hit point. If no hit point is identified, repeat the same procedure to the selected quadrant.

The quadrant search heuristics consists of *Tile Partition* and *Block Scan Algorithm*. Intended to be guided by the triangle gravity center, the tile partition implies the recursive partition of the tile into four identical quadrants and the selection of one of the four quadrant's center pixel to detect if the pixel is inside the triangle. The partition and testing process continues until the block is small enough to be scanned in a pixel-by-pixel manner.

C.1 Tile Partition

Different from testing the triangle vertices and triangle gravity center, tile partition tests certain pixels in the current processing tile instead of judging if a point belonging to the triangle is inside the tile. The tile partition can be stated as follows:
1. Test the center pixel of the current quadrant (at the first iteration the current quadrant is the current processing tile) to detect if the point is inside the triangle;
2. If the pixel is inside the triangle a hit was found and process is stopped;
3. If the center pixel of the current processing tile is not inside the triangle, the current quadrant is partitioned into

four identical quadrants. The quadrant that extends to the direction of triangle gravity center is selected;
4. If the chosen quadrant is small enough, the algorithm is stopped (the Block Scan is initiated), otherwise go to $STEP$ 1.

To test if a pixel is inside a triangle, the linear edge function algorithm, discussed in Section II, is utilized, associated with the OpenGL point sampling rule [5].

In order to make things clear, the tile partition algorithm is illustrated by an example, shown in Figure 3. As seen in the figure, the center pixel of the current processing tile is the point $P1$. Since $P1$ is not in the region of the triangle $\triangle ABC$ the tile is divided into four identical quadrants, $Q1$, $Q2$, $Q3$ and $Q4$. The quadrant $Q1$ is selected as it is closer to the triangle gravity center of the $\triangle ABC$ lying in the extension area of $Q1$. Again, the center pixel, $P2$, of $Q1$ has to be checked. Because $P2$ is still outside the triangle, quadrant $Q1$ is partitioned into four pieces, $Q1\_1$, $Q1\_2$, $Q1\_3$ and $Q1\_4$, and $Q1\_1$ is selected and its center pixel $P3$ is evaluated. $P3$ is the first valid pixel encountered in the process of partitioning and testing so a hit point was located and the coordinates of $P3$ are stored in a register.

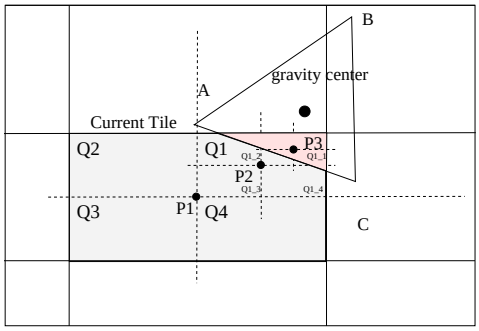
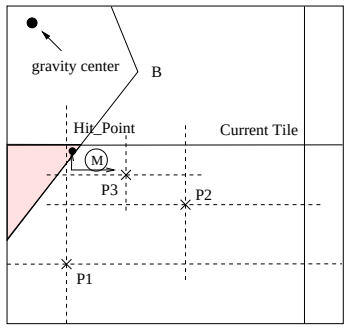


Fig. 3. An example of Tile Partition Algorithm.

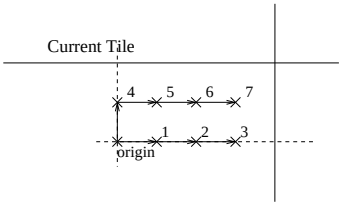
C.2 Block Scan Algorithm

In case that the current processing tile has been partitioned into quadrants that are small enough whereas a hit point has not been located, the quadrant can be scanned incrementally in a pixel-by-pixel manner. The *Block Scan* algorithm is meant to locate a hit point inside the "small" quadrant. The meaning of "small" depends on the size of the tile defined. Normally, it means that the block constructed by pixels is too small to have a real center pixel, see Figure 4(b). In this figure, the block covers 8 pixels. Pixel 1,2,5 and 6 construct the central area of the block, yet it is hard to tell which pixel is the real center of the block. In this case such a block, called *last block* in the reminder of this report, is considered to be small and passed through the Block Scan process.

Figure 4(a) illustrate an example of the block scan process. According to the figure, the tile partition didn't locate a hit point from point $P1$, $P2$ and $P3$. Moreover the position of triangle gravity center suggests the triangle probably lies left-top direction of the point $P3$. Consequently, block M is selected for block scan in order to detect if a hit point exists in its area. The scanning trajectory is depicted in Figure 4(b). As shown in Figure 4(b), the traversal always starts from the left-bottom corner, *origin* in the figure, of the last block. The bottom line is scanned completely before switching the traversal to the top line. To evaluate pixels on the trajectory, the same method, as the Tile Partition uses, is employed.



(a) Block_scan



(b) Block_scan_trajectory

Fig. 4. Block scan process.

D. Backup Strategy

Some pathological triangles, for example the one depicted in the Figure 5, contribute fragment attribute values to the current processing tile, but the hit point can never be located by the algorithms discussed in the previous sections. To reach the triangle, quadrant search intends to track to the triangle along the direction of the triangle gravity center. However, in case of the one in Figure 5, the hit point cannot be located, for the triangle intersects the tile at area N. Therefore, it is necessary to derive some backup strategies to guarantee that a hit point is located also in the case of such triangles.

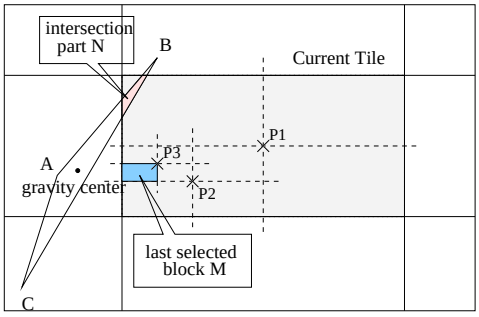


Fig. 5. Pathological triangle case example 1.

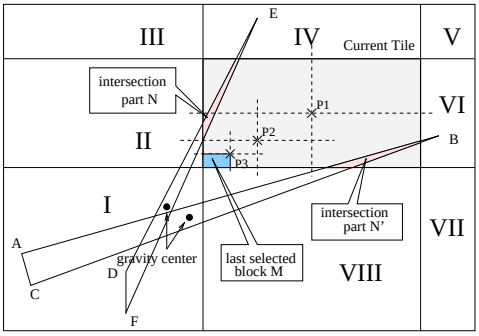


Fig. 6. Pathological triangle case example 2.

TABLE I
BORDER(S) TO BE SCANNED FOR EACH KIND OF PATHOLOGIC TRIANGLE.

Region	I	II	III	IV
Border	LB	L	LT	T
Region	V	VI	VII	VIII
Border	RT	R	RB	B

Analyzing the case shown in Figure 5, we find that when the quadrant search strategy fails to find a hit point, the hit point position can be located by scanning the border to which the gravity center is closer. For instance, as shown in Figure 5, the left border of the current processing tile is adjacent to the gravity center, and accordingly the hit point can be found by visiting every pixel on the left border of the tile.

However another cases should also be taken into account. In Figure 6, the outer area of the tile is divided into eight area, named in order from I to VIII. If the triangle gravity center falls into one of the region II, IV, VI and VIII, scanning the adjacent border is adequate to find the hit point. Yet scanning only one border is not enough, if the gravity center lies in the region I, III, V, VII. For instance, in Figure 6 there are two triangles, whose gravity centers fall in the area I. The Quadrant Searches Heuristics fails to locate the hit point after scanning the last block. Backup strategy is initialed to search the hit point. Both the left and the bottom border are required to be scanned due to the two possible intersection cases. Table I presents the border(s) need to be scanned, when a pathologic triangle occurs. In the table, letter L, R, T and B are respectively representing the left border, right border, top border and bottom border. The Region item suggests the area that gravity center falls in. Clearly, the pathologic triangles introduce many computational overhead when attempted to locate the hit point of triangles.

IV. EXPERIMENTAL RESULTS

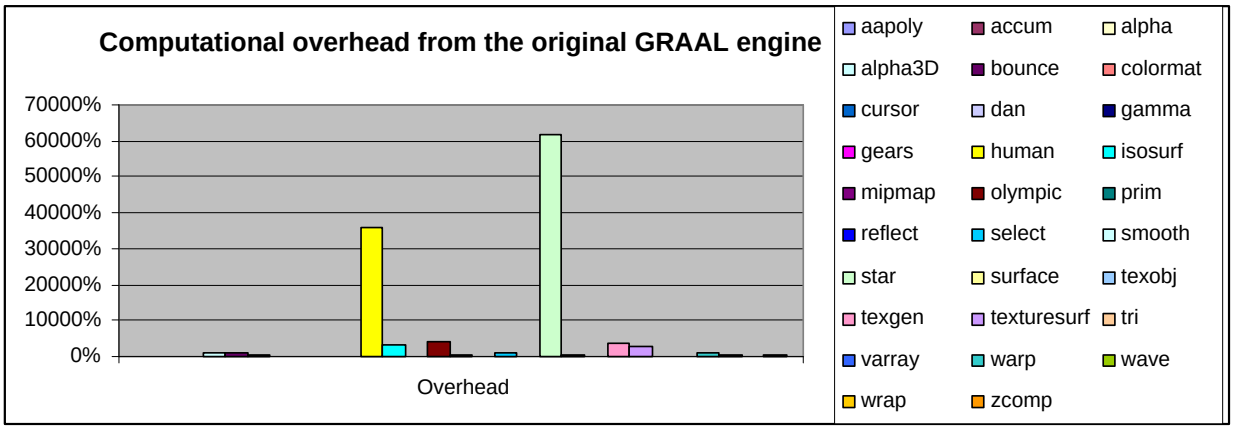
To evaluate the effectiveness of the heuristic algorithms discussed in Section III, we have modelled the algorithms in SystemC at RT-level. Since the functionality of the algorithms is to locate a hit point in a triangle, we named the functional unit as Hit Point Functional Unit, abbreviated as HPFU. In this section, we present the experimental simulation results, after integrating HPFU into the GRAAL rasterization software/hardware co-simulator [2]. Furthermore, these results are compared to the performance of the exhaustive algorithm currently adopted by the GRAAL engine.

30 OpenGL applications have been selected as benchmarks for our simulation. The benchmark suite includes some general cases, such as the “aapoly” application from [5], and some extreme cases, such as AWadvs-04 component of the SpecViewperf 6.12 benchmark [6]. During the process of experiments, a set of counters have been recorded. The performance of the original GRAAL rasterizer and the modified one with embedded HPFU is evaluated according to these counters. The counters recorded are as follows:

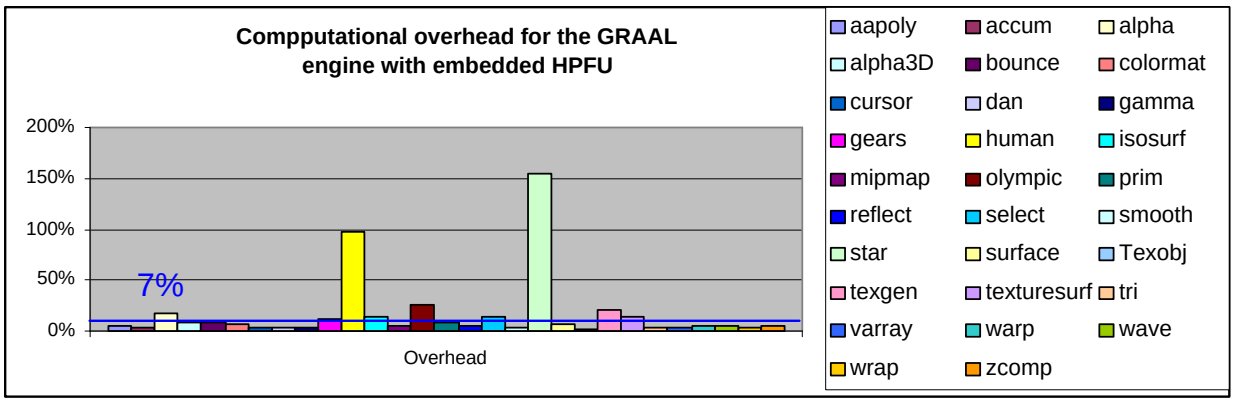
- fragment_in_triangle: recording the number of valid pixels in a triangle;
- fragment_hit_and_in_tri: recording the number of pixels visited from the starting point to a hit point in a triangle, plus the number of valid pixels in a triangle;
- cycle_tested_hit: the number of cycles for locating a hit point. This counter is used when the performance of the new rasterizer is evaluated.

A. Overhead comparison

We have simulated the 30 benchmarks on the modified engine that embeds HPFU against the original rasterizer, aiming at finding out the computational overhead cost.
The number of clock cycles spent by the original GRAAL engine before locating a hit point, is represented by the



(a) Computational overhead for the original GRAAL.



(b) Computational overhead for GRAAL with embedded HPFU.

Fig. 7. Computational overhead comparison.

following formula:

$$cycles\_hit = (fragment\_hit\_and\_in\_tri - fragment\_in\_triangle) \times cycles\_for\_each\_iteration \quad (4)$$

where, the factor, $cycles\_for\_each\_iteration$, represents clock cycles that are consumed on testing if one pixel is inside the triangle. Referring to [1], 4 clock cycles are spent on testing a pixel. On the other hand, the clock cycles consumed by the modified engine before locating a hit point is represented by the counter, $cycle\_tested\_hit$.

The number of clock cycles spent computing fragment values is calculated by the formula as follows:

$$cycles\_Valid\_Pixel = fragment\_in\_triangle \times cycles\_compute\_fragment \quad (5)$$

where the $cycles\_compute\_fragment$ is the clock cycles cost on computing a pixel's fragment attribute values, which is 5 [1]. Consequently, the computational overhead for one benchmark generated by the original engine is computed by formula:

$$overhead1 = cycles\_hit / cycles\_Valid\_Pixel \quad (6)$$

while the overhead introduced by the modified engine is computed by formula:

$$overhead2 = cycles\_tested\_hit/cycles\_Valid\_Pixel \quad (7)$$

The average computational overhead for locating a hit point involved in the original rasterization engine is computed by formula below:

$$Avg.overhead1 = \Sigma cycles\_hit/\Sigma cycles\_Valid\_Pixel \quad (8)$$

while the average overhead involved in the modified engine is computed by formula:

$$Avg.overhead2 = \Sigma cycles\_tested\_hit/\Sigma cycles\_Valid\_Pixel \quad (9)$$

Figure 7(a) and Figure 7(b) are respectively presenting the computational overhead generated by the original GRAAL and the modified one and the improvement can be observed. An overhead of 3%–155% with an average of 7% indicates a significant reduction in the computational overhead to locate a hit point in a triangle, compared with the average of 1141% from the original GRAAL rasterization engine.

The proposed heuristics were implemented as a functional unit (HPFU) at RT-level in SystemC modeling language and synthesized employing Synopsys tools in a commercial 0.18μm process technology. The integration into the GRAAL rasterization engine resulted in an area increase of about 10% percent.

V. Conclusions

This paper presented a series of hardware algorithms to reduce the computational overhead to locate the first rasterization tile position inside the primitive to be rasterized when the tile-based rasterization adopts the classical primitive traversal algorithm. These algorithms can be applied sequentially in a simple-to-complex order for searching a suitable starting tile rasterization position inside the primitive as follows: check if any of the vertices is in the tile, check if the triangle center of gravity (COG) is in the tile, recursive tile quadrant division based on COG attractors, and partial tile boundary scan. The algorithms were modeled in SystemC at the RT-level and integrated in a full-fledged OpenGL-compliant hardware rasterizer SystemC model. Simulation results on a benchmark suite consisting of 30 OpenGL applications have indicated that the throughput penalty is reduced to about 7% at the expense of about 10% increase in the hardware area when the entire OpenGL-compliant hardware rasterizer is synthesized in a commercial 0.18μm process technology.

References

[1] D. Crisu, S. D. Cotofana, and S. Vassiliadis. A Proposal of a Tile-Based OpenGL Compliant Rasterization Engine. Technical report, Delft University of Technology, January 2002.

[2] D. Crisu, S. D. Cotofana, S. Vassiliadis, and P. Liuha. GRAAL - A Development Framework for Embedded Graphics Accelerators. In *Proceedings of Design, Automation and Test in Europe (DATE'04)*, pages 1366–1367, February 2004.

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

[4] Tomas Moller and Eric Haines. *Real-Time Rendering*. A K Peters, 1999.

[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] SPECviewperf 6.1.2, URL: http://www.specbench.org.