On the Hardware Implementation of Triangle Traversal Algorithms for Graphics Processing

Pablo Royer Pablo Ituero Marisa López-Vallejo Carlos A. López Barrio
Dpto. de Ingeniería Electrónica, ETSI Telecomunicación, Universidad Politécnica de Madrid
Ciudad Universitaria s/n, 28040 Madrid, Spain
{proyer,pituero,marisa,barrio}@die.upm.es

***Abstract*—Current GPU architectures provide impressive processing rates in graphical applications because of their specialized graphics pipeline. However, little attention has been paid to the analysis and study of different hardware architectures to implement specific pipeline stages. In this work we have identified one of the key stages in the graphics pipeline, the triangle traversal procedure, and we have implemented three different algorithms in hardware: bounding-box, zig-zag and Hilbert curve-based. The experimental results show that important area-performance trade-offs can be met when implementing key image processing algorithms in hardware.**

I. Introduction

The video game and interactive entertainment industry is getting revenues of tens of billions of dollars and increasing every year. The heart of the technological developments that makes possible to keep satisfied the users increasing demand is the Graphics Processing Unit (GPU). Furthermore, the computational power and programmability of today's graphics hardware can be applied to many areas like computer graphics, film rendering, physical simulation, and visualization.

GPUs implement basic graphical operations optimized for graphics processing. The 3D graphics computations are organized into a graphics pipeline that performs a series of computation stages to go from a 3D model to the pixels on a monitor. The GPU processes and transforms the input vertexes into screen-space geometry, which in turn is divided into pixel-sized fragments, in a process called rasterization, according to which pixels are covered by that geometry. Each fragment is then associated with a pixel position on the screen. Finally, the fragments are processed and assembled into an image made of pixels [1].

Figure 1 shows a conventional graphics pipeline, which contains around a dozen stages. The input vertex stream passes through a computation stage that transforms and computes some of the vertex attributes generating a stream of transformed vertexes. The stream of transformed vertexes is assembled into a stream of triangles, each triangle keeping the attributes of its three vertexes. After that, the stream of triangles may pass through a stage that performs a clipping test. Then each triangle passes through a rasterizer that generates a stream of *fragments*, discrete portions of the triangle surface that correspond to the pixels of the rendered image. Fragment attributes are derived from the triangle vertex attributes. This stream of fragments may pass through a number of stages performing a number of visibility tests (stencil, depth, alpha and scissor) that will remove non visible fragments and then will pass through a second computation stage. The fragment computation stage may modify the fragment attributes using additional information from n-dimensional arrays stored in memory (*textures*). Textures may not be accessed as stream. The stream of shaded fragments will, finally, update the frame-buffer [2].

This work was funded by the CICYT project DELTA TEC2009-08589 of the Spanish Ministry of Science and Innovation.

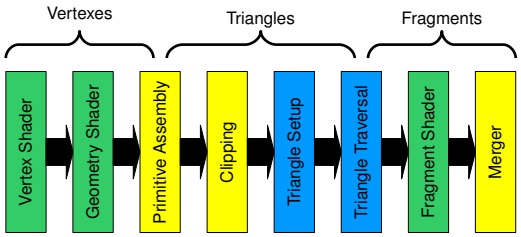

Fig. 1. Graphics pipeline.

The typical input to the pipeline is tens to hundreds of thousands of vertexes that define the geometry of the scene. The basic elements are made up of triangles in general, since triangles can be described with three vertexes, and only reside in one plane. Thus a key stage in the graphics pipeline is the one that maps each pixel to a given triangle. To accomplish this task triangle traversal algorithms are required, whose computation is responsible for a significant performance overhead due to the huge number of vertexes which are processed. The hardware implementation of the triangle traversal should be carefully analyzed in order to minimize the performance penalty incurred by this stage. This is the objective of the work we present here. Very few previous approaches have dealt with the hardware implementation of triangle traversal algorithms. Moreover, the way how triangle traversal is actually implemented in GPUs is unknown, since most work around GPUs is kept secret because of high competitiveness in the hardware segment of this industry. In [3] a specific full-custom implementation for a tile-based triangle traversal algorithm is studied, but paying special attention to power minimization. Other works focus on the quantization impact on the quality of the resulting images [4].

The main contributions of this work are the following:

- To the best of our knowledge this is the first time that the hardware implementation of different triangle traversal algorithms has been analyzed.
- The quantization of each involved signal has been studied.
- An in-depth analysis of the implications of each algorithm concerning throughput and latency has been performed.

Next, the basis of triangle traversal and depth interpolation using edge functions are introduced. The following sections explain the hardware implementation of the three algorithms. Finally the results are detailed and some conclusions are drawn.

II. Triangle Traversal Overview

The triangle traversal stage checks whether each of the pixels is inside the triangle or not. For each pixel (or sample point) that overlaps the triangle, a fragment is generated (see figure 2). Each

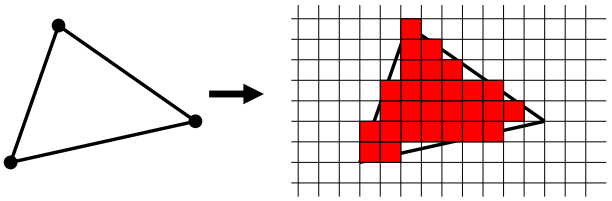

Fig. 2. Triangle traversal: for each pixel inside the triangle, a fragment is generated. [5]

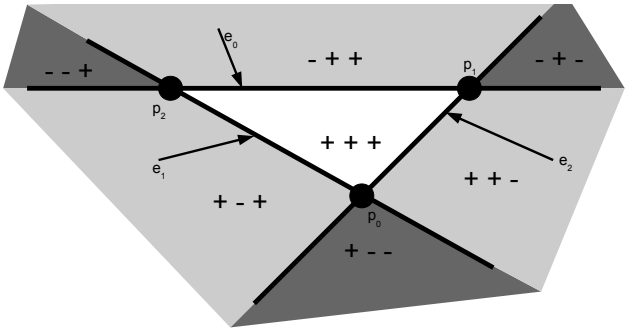

Fig. 3. Regions defined by the Edge Equations of a triangle.

fragment has information about its location in the screen, its location in the triangle (how far it is from the vertexes) and its depth [5] [6].

A triangle is defined by its three vertexes (v_0, v_1 and v_2), and each vertex is composed by three coordinates: $v_i = (x_i, y_i, z_i)$ where x_i and y_i are the location of the vertex in the screen and z_i is the depth of the vertex.

In a triangle, we can define three edge equations (e_i with $i = \{0, 1, 2\}$), one for each couple of vertexes. The edge e_i is the one defined by the vertexes opposite to v_i. Checking in which side of each edge a pixel is, we can decide whether the pixel is inside the triangle or not.

A. Edge Equations

An edge function is the equation of the line between two vertexes of the triangle, whose implicit general equation is:

$$e(x, y) = ax + by + c = 0 \tag{1}$$

where x and y are the integer coordinates of the center of the pixel[1]. The a, b and c coefficients are computed as:

$$\begin{aligned} a &= y_p - y_q \\ b &= x_q - x_p \\ c &= x_p y_q - x_q y_p \end{aligned} \tag{2}$$

where the p and q sub-indexes denote the two vertexes that define the edge.

The sign of e_i tells us if a point is on one side of the edge or on the other. The region inside the triangle corresponds to the region where the three edge equations are greater than zero. Any region outside has at least one edge equation result less than zero, as shown in figure 3. The triangle traversal using edge equations is well explained in [7] and [8].

Pixels that are located exactly on the edge of the triangle ($e(x, y) = 0$) may belong or not to the triangle. Considering two triangles sharing an edge, a pixel that lies on the shared edge should belong to one and only one of the two triangles, otherwise this can generate incorrect results. Given that for two triangles sharing an edge, the shared edge equation coefficients are negations of one another, a tie-break rule [9] can be established as it is shown in Table I.

[1]In this work no anti-aliasing has been considered so only one sample per pixel is taken.

TABLE I
TIE-BREAK RULE FOR DETERMINING IF A POINT IS INSIDE AN EDGE. [9]

$e_i(x, y)$	a_i	b_i	inside
> 0			true
< 0			false
$= 0$	> 0		true
$= 0$	< 0		false
$= 0$	$= 0$	≥ 0	true
$= 0$	$= 0$	< 0	false

B. Incremental Traversal

Computing the e_i values of a fragment requires six multiplications and six additions. But, seeing that:

$$\begin{aligned} e_i(x+1, y) &= e_i(x, y) + a_i \\ e_i(x-1, y) &= e_i(x, y) - a_i \\ e_i(x, y+1) &= e_i(x, y) + b_i \\ e_i(x, y-1) &= e_i(x, y) - b_i \end{aligned} \tag{3}$$

given the edge functions values of a fragment, its four neighbours can be tested performing only three additions (one per edge) instead of performing again six multiplications and six additions. That is the reason why most traversal algorithms incrementally explore the triangle pixel by pixel.

C. Depth Interpolation

After the triangle traversal is done, it is known if a pixel belongs or not to the triangle, but there is more information that has to be computed. Usually the vertex has a depth coordinate, colour or texture, thus the generated fragments need this information, which is usually calculated by interpolating the values at the vertexes.

When traversing the triangle using edge equations, the depth coordinate can be interpolated with a similar equation whose coefficients are derived from the edge equation coefficients as:

$$\begin{aligned} a_z &= \frac{a_0 z_0 + a_1 z_1 + a_2 z_2}{\sum c_i} \\ b_z &= \frac{b_0 z_0 + b_1 z_1 + b_2 z_2}{\sum c_i} \\ c_z &= \frac{c_0 z_0 + c_1 z_1 + c_2 z_2}{\sum c_i} \end{aligned} \tag{4}$$

where a_i, b_i and c_i are the coefficients of the edge equation and z_i are the depths of the three vertexes. The a_z, b_z and c_z coefficients are calculated only once per triangle as explained in [8].

The depth value of a vertex, denoted as e_z is computed from the previous coefficients as:

$$e_z = a_z x + b_z y + c_z \tag{5}$$

It can also be calculated incrementally along with the e_i values of the edges.

D. Bit Length Considerations

In our work we consider a maximum screen resolution of 4096×4096 pixels (as [2]) so 12 bits are required to represent the x and y coordinates as an unsigned integer. From how a_i and b_i coefficients are calculated (see [8]) a 13-bit signed integer is needed. For the same reason c_i coefficients are represented as 25-bit signed integers while the e_i values require a 27-bit signed integer. The sign bit of e_i values is only required while traversing the triangle, but the value

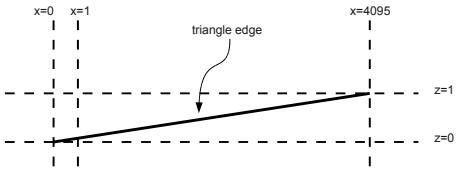

Fig. 4. Determining fractional parts of the a_z and b_z coefficients.

TABLE II
BIT-WIDTH OF THE VARIABLES.

variable	signed	bits
x_i, y_i	no	12
z_i	no	24
a_i, b_i	yes	13
c_i	yes	25
$e_i(x, y)$	yes	27
a_z, b_z, e_z	-	$36 = (24 + 12)$

that is output with a fragment does not need that bit since the value is positive because the pixel is inside the triangle.

In order to avoid artifacts in the rendered frame, at least 24 bits are required to represent the depth coordinate [10]. Therefore it has been represented as a 24-bit unsigned integer and the interpolated value will also be a 24-bit integer. While the e_i values need to have an appropriate value for pixels outside the triangle, the interpolated depth value —e_z— is only taken into account for pixels that are inside so it does not matter if the value overflows as it will recover an appropriate value when traversing back to an internal pixel. As a consequence, only 24 bits are required to represent the integer part of e_z, a_z and b_z coefficients; higher bits —including the sign bit— do not need to be represented.

Another difference between a_i or b_i and a_z or b_z is that the former are integers whereas, due to the division, the latter are not. To decide how many decimal bits are necessary to represent a_z and b_z we will consider the worst case, that is a triangle where the z value is increased as little as possible while the triangle is traversed. This case is shown in figure 4. From the figure it can be easily understood that the lowest value that a a_z or b_z coefficient will take is $\frac{1}{4096}$. To represent this, 12 fractional bits are required. So the e_z, a_z and b_z are represented as 36 bit-width numbers, with a 24-bit integer part and a 12-bit fractional part. The fractional part is used during the traversal while only the rounded integer part is output as a fragment information.

This quantization has been obtained considering the worst case for triangles that are not realistic, we leave as future work the study of the quantization noise produced by decreasing the number of bits allocated to the fractional part of the depth interpolation coefficients.

All the bit lengths from the resulting parameters are summarized in Table II.

III. Triangle Traversal Algorithms

A. Bounding-Box Traversal

The bounding-box traversal algorithm tests every pixel that is inside the smallest rectangle that contains the triangle. As the triangle is fully inside the rectangle, every pixel belonging to the triangle will be tested and a fragment will be generated.

This algorithm explores the bounding-box line by line. To enable every new pixel edge equations to be calculated from a previous

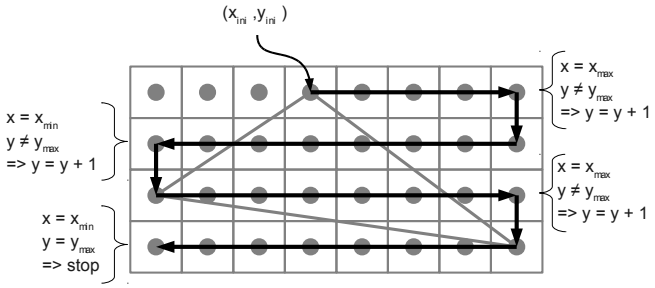

Fig. 5. Bounding-box Traversal Algorithm.

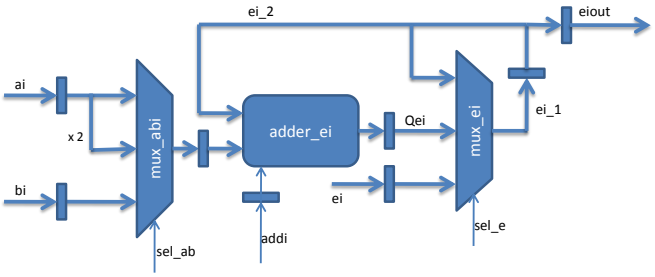

Fig. 6. Full schematic of an e_i adder.

neighbour pixel, the order in which a line is traversed is changed for every new line.

While the e_i values —which indicate if the pixel is inside or not— are updated, the x and y coordinates —that indicate the location in the screen— are also updated.

The algorithm control is performed by comparing the x and y coordinates with the x_{min}, x_{max} and y_{max} values provided by the set-up stage[2]. The first pixel tested is the one located at (x_{ini}, y_{ini}) where $y_{ini} = y_{min}$. This process is explained by figure 5.

1) Adder/subtracter Implementation: The main module that composes the bounding-box traversal stage is an adder / subtracter, which performs the addition or subtraction of the a_i or b_i coefficients to the current e_i value. The result is the new e_i value that replaces the previous one. The implementation is shown in figure 6.

In order to reduce the delay of the hardware, two registers have been introduced in the loop that composes the adder/subtracter. To avoid bubbles in the pipeline it is necessary to enable the addition of $2a_i$ that makes the box to be traversed by two pixels to the right or the left. With this solution bubbles are avoided during a line traversal but cannot be avoided at the beginning and the end of the line.

The implementation of the x and y adders is quite similar to the e_i implementation, but includes the comparators required to change the scanning direction.

2) Generating the Inside Flag: This is done by testing the e_i, a_i and b_i values as explained in table I. However comparing them with zero and using the results to decide if the pixel is inside would introduce a too long delay. Thus, the process is split in two steps: on the first one, the coefficients are checked if they are greater, equal or

[2]The previous stage in the graphics pipeline that is in charge of computing the traversal coefficients and the bounds of the box.

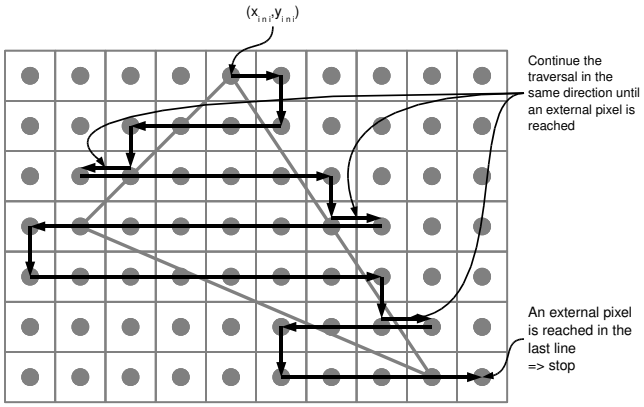

Fig. 7. Zig-zag traversal algorithm.

lower than zero, this generates 2-bit wide signals easier to process than long bit-width e_i, a_i and b_i values. On the second step these signals are combined following Table I to find out if the pixel is on the positive side of the edge.

B. Zig-Zag Traversal

As the bounding-box traversal, the zig-zag traversal tests the pixels of a triangle line by line, but the length of the line differs from one another. Only the pixels that are useful to ensure that no pixel belonging to the triangle is being left have to be visited. The process is shown in figure 7.

In most of the scan lines the process is quite simple. The first pixel location to be checked is the one just under the last pixel of the previous line. If this pixel is not outside the triangle, the line is explored in the same direction as the previous one until an external pixel is reached. Then the line is explored in the opposite direction, traversing the triangle from a pixel that is outside to another that is also external but in the other side of the triangle.

However, there are some triangles, usually thin ones, for which this simple process can lead to mistakes because two edges can be crossed in a single step. To avoid this, instead of testing only whether the pixel is inside the triangle or not, the three edge equations results are checked separately. In this way we can know for a pixel that is outside the triangle, on which side it is. We denote the place of the screen where the pixel is with a triplet $S = (s_0, s_1, s_2)$, where s_i is the sign of the edge equation e_i. If the current pixel is in the positive side of the edge i, then $s_i = 1$ otherwise $s_i = 0$.

1) Inside Flag: The pixel is now said to be in the positive side of an edge if its e_i value is greater than or equal to zero, while a_i and b_i are not tested anymore. The difference between this algorithm and the former is that pixels located just in the edge ($e_i = 0$) will always be considered as belonging to the triangle while those whose a_i is negative are actually outside. This does not lead to mistakes as the other test is still executed to decide if a fragment is generated or not. On the other hand, none of the pixels that are inside will be omitted by this new test.

2) Control Signals: As seen before the traversal algorithm needs information about the location of the current pixel in relation to the triangle, this information was called $S = (s_0, s_1, s_2)$. When the traversal of a line is finished the S value at the last pixel is stored as S_{stored} and the screen is traversed one pixel down. The new line is traversed in the same direction until a pixel that is outside in the same side of the triangle is reached. This corresponds to a pixel whose S value has zeros at least in the same places as the S_{stored} value:

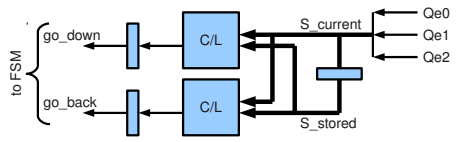

Fig. 8. Zig-zag traversal control signals generation.

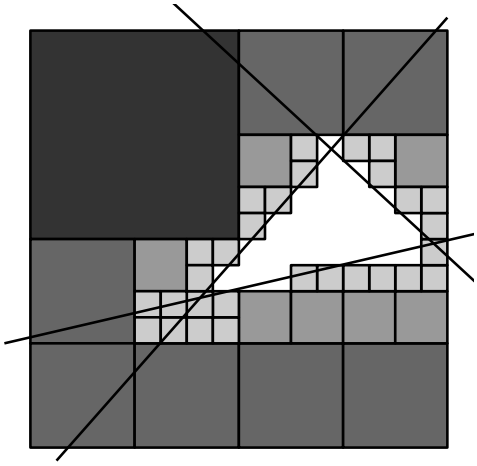

Fig. 9. Triangle traversal using Hilbert curve.

$$\overline{S_{current}} \cup S_{stored} = (1, 1, 1) \quad (6)$$

Once we are placed in the required side of the triangle, the actual traversal can start in the opposite direction until a pixel that is in the opposite side of the triangle is reached. This corresponds to a pixel whose S value has ones at least in the same places where S_{stored} had zeros, and that is not an internal pixel whose $S = (1, 1, 1)$:

$$(S_{current} \neq (1, 1, 1)) \cap ((S_{current} \cup S_{stored}) = (1, 1, 1)) \quad (7)$$

The additional hardware for this function is shown in figure 8.

C. Hilbert Curve Traversal

The Hilbert curve traversal is a different approach than bounding-box or zig-zag traversal. Instead of traversing the triangles pixel by pixel, the screen is divided into tiles of different sizes: if a tile overlaps the triangle it is divided into four smaller sub-tiles. The triangle is traversed repeating this same process for each tile and sub-tile until a one pixel size tile is achieved [9].

In this algorithm, the next pixel being tested is not always the neighbour of the current one so a multiplication is required. Fortunately the jump between a pixel and the next one is always a power of two: 2^n, so the multiplication is easily achieved by shifting the a_i or b_i coefficients by n bits.

Figure 9 shows how a triangle is traversed: different color rectangles represent tiles that are discarded without subdividing, the darker a tile is the less times it has been subdivided before discarding it. White pixels belong to the triangle and need to be subdivided until the smaller tile size to generate a fragment.

1) Four Corners Test: As the Hilbert curve traversal is a tiled traversal, one of the core operations of the algorithm is to test the four corners of the tile to decide whether it has to be subdivided or not. If the four corners are in the negative side of at least one of

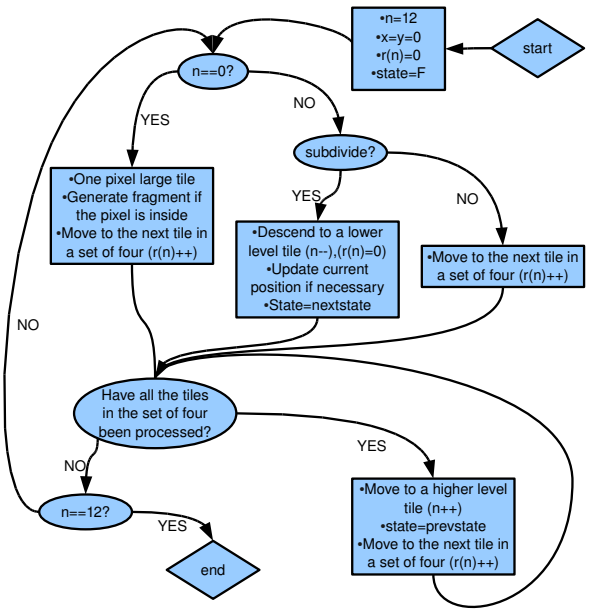

Fig. 10. Hilbert curve triangle traversal algorithm.

the edges, the tile does not overlap the triangle. If the four corners are outside the triangle but not for the same edge is not enough to say that the tile does not overlap the triangle and the tile has to be subdivided [8].

The first operation to be performed is $e_i + a_i$. Then, $e_i + b_i$ is executed taking advantage of the pipeline as this operation does not require the result of the previous one. With those operations the next two corners are tested. The results of $e_i + a_i$ are stored as the current e_i values while the results of $e_i + b_i$ are not. This way the fourth corner can be tested as $(e_i + a_i) + b_i$ and we can come back to the first corner by $(e_i + a_i) - a_i$. This last operation is required because the result of $e_i + a_i$ had been loaded as the current e_i values and we need to recover the initial position after testing the four corners.

2) Additional Hardware Modules: This implementation needs new hardware modules not required by the previous ones. The size of the tile that is currently being processed is denoted as n, that means that the tile is $2^n \times 2^n$ pixels. This variable is stored in a register and an adder/subtracter changes the values according to the current tile size. As the n value is updated the a_i and b_i coefficients are shifted to the corresponding direction to make them fit to the appropriate tile size.

When subdividing a tile, four sub-tiles are generated. In order to distinguish those four tiles an r value ranging from 0 to 3 is set. As any of those sub-tiles can be further subdivided, a new r value is needed for the lower level tiles. We denote these values as $r(n)$, they are stored in a 13-word $\times$ 2-bit register.

When a tile is subdivided the order in which the four sub-tiles are explored is determined by a recursive algorithm that generates a Hilbert curve. This order is denoted as a high level state represented by two bits. The next order when going down to a lower level tile depends on the current order and the current r value. Meanwhile the order when moving to a higher level tile is recovered from the current order and the most significant bit of the x and y coordinates. Those functions are implemented employing combinational circuits with a 4-bit input and a 2-bit output.

The traversal process for the Hilbert curve method that we have implemented is shown in figure 10.

TABLE III
AREA AND DELAY OF THE THREE IMPLEMENTATIONS

	Bounding-Box	Zig-zag	Hilbert
Slices	375	483	777
Delay	5347 ps	4625 ps	6818 ps
Frequency	187 MHz	216 MHz	147 MHz

IV. RESULTS

The three algorithms functionality has been validated by Matlab simulation. Then the algorithms have been described in VHDL and simulated using ModelSim. The simulation results have been compared to the ones obtained with Matlab to check they were correct.

As test-benches, in addition to random triangles, we have used special triangles that could lead to mistakes if the hardware was no properly designed. Those triangles were the ones located at the corners and the edges of the screen and also special-shape triangles that were very thin or small.

A. Area and Speed Results

In order to obtain comparable area and speed results, we have implemented the three triangle traversal modules on a Xilinx XC2VP30 Virtex-II Pro FPGA using Xilinx ISE 9.2i. It must be said that the natural target for a GPU is direct implementation on silicon, so metrics taken from an FPGA synthesis should be taken as a coarse indication of the comparison in a more realistic situation. However, some research groups show interest in implementing GPU related modules on them [11]. The design has been synthesized with a speed optimization goal and with the register balancing option enabled. The Map and Place and Route processes have been done with a speed reduction strategy and a multi-pass place and route.

The results of the FPGA implementation related to speed and delay are shown in Table III. We can see that, as expected, the Bounding-Box algorithm is the one that requires the least area while the Zig-zag algorithm is the one that can work at the highest speed: 216 MHz. The Hilbert curve traversal is the algorithm that takes the highest area and that runs at the lowest speed.

B. Throughput

1) Bounding-Box: The Bounding-Box triangle traversal achieves almost a rate of one pixel test per clock cycle, but due to the pipeline, this rate cannot be maintained when the traversal direction changes. As a consequence it takes $n + 3$ clock cycles to traverse an n-pixels scan-line. Therefore the time required to traverse a triangle depends on the size of its bounding-box: testing all the pixels of an $n \times m$ rectangle lasts $m \cdot (n + 3)$ clock cycles.

The area of the biggest triangle that can be contained on a rectangle is half the area of the rectangle. So, considering a big enough bounding-box, the highest rate that can be achieved with this algorithm is two clock cycles per fragment generated. Employing the figures from our FPGA implementation, that corresponds to a throughput of $93.5 \cdot 10^6$ fragments/second.

2) Zig-zag: As the Bounding-box, the Zig-zag traversal tests one pixel per clock cycle, but some extra cycles are required when the traversal direction changes. Some other cycles are also needed when seeking external pixels where the scanning has to begin. So both the number of cycles required to traverse a triangle and the rate of pixel tested against the pixel that actually belongs to the triangle will depend on the triangle shape.

TABLE IV
THROUGHPUT AND LATENCY OF THE THREE IMPLEMENTATIONS

	Bounding-Box	Zig-zag	Hilbert
Max. throughput	$93.5 \cdot 10^6$ fr./s	$194.6 \cdot 10^6$ fr./s	$13.6 \cdot 10^6$ fr./s
Min. latency	7 cycles	10 cycles	649 cycles

The bigger a triangle is the better rate will be achieved as the number of extra clock cycles will be masked by many more fragments generated. In the same way triangles whose edges are not near horizontal will have a better rate as less cycles will be missed searching for an external pixel when starting a new line. On the whole, rates above 90% can be achieved for triangles larger than 2000 pixels, this rate corresponds to a maximum throughput of $194.6 \cdot 10^6$ fragments/second in our FPGA implementation.

3) Hilbert Curve: The Hilbert curve triangle traversal is the one that requires more cycles to generate fragments. The reason behind this is that when generating two consecutive fragments from a single triangle it has to move to at least one higher level tile and then move back to the next lower level tile. This means many operations to move from a tile to another and to test if the tile overlaps the triangle.

The number of cycles that this algorithm takes to explore a triangle strongly depends on the triangle shape, but we can estimate a higher bound. It takes 43 clock cycles to explore the four pixels of the lower level tile, so we cannot achieve a rate better than 10.75 cycles per fragment generated that corresponds to a throughput of $13.6 \cdot 10^6$ fragments/second in our FPGA implementation.

C. Latency

The minimum latency of the modules has been simulated for the smallest triangle, that had a half-pixel area. The lowest latency from the different orientations and positions of the triangle has been kept. The zig-zag traversal has higher latency than the bounding-box as it searches for external pixels that may be outside the box. The Hilbert curve traversal has the highest latency as even if the triangle leads to one or even none fragment generated, we have to go down through at least twelve levels of tiles and then go back. The throughput and latency of the three modules implementations are shown in Table IV.

D. Discussion

According to the results, the Hilbert curve traversal has worse performance in area and throughout than the two other algorithms, but it has an advantage: the pixels are generated in a spatially coherent order [9], better that Bounding-Box or Zig-zag whose pixels are generated line by line. This will improve the data locality of the texture caches decreasing the number of times a same texture needs to be loaded. The high latency disadvantage of the Hilbert curve traversal could be solved using different rasterizers depending on the triangle size as suggested by [12].

For the two other algorithms, the order in which the pixels are generated is the same for both of them. The zig-zag traversal seems to be a better approach: even if it has a larger area, on average fewer external pixels are visited and so its throughput is much higher.

A different implementation of the Hilbert curve traversal could use pipelined stages to process tiles of different sizes, or perform more operations per clock cycle at the cost of a larger area. Another improvement could be a hybrid implementation using a fixed pipelined Hilbert curve traversal down to a certain tile size level and then using one of the other algorithms to test the pixels inside the tile. This will enable a speed close to the Zig-zag or Bounding-Box traversal and a lower latency than the Hilbert curve traversal, keeping the spatial coherence of the generated fragments that a tiled traversal achieves. This is left as future work.

V. CONCLUSION

The GPU field has undergone a hardware revolution during the last fifteen years attaining amazing processing rates far beyond general-purpose CPUs. This has been made possible thanks to a very deep pipeline in which each of the stages is very carefully designed. This work has analyzed the trade-offs of the hardware implementation of the triangle traversal stage.

Even though FPGAs are not the most suitable platform for GPU implementation, the results related to latency and number of pixels checked or fragments generated per cycle will be the same. The area and delay, and as a consequence the throughput, will strongly depend on the hardware, but differences from an algorithm to another should remain similar.

Specifically, three different algorithms have been implemented and compared: bounding-box, zig-zag and Hilbert curve-based. Our results set a compromise between area, frequency, throughput and latency that can be used by future designers.

REFERENCES

[1] J. D. Owens, "GPUs tapped for general computing," *EE Times*, 13 Dec. 2004. [Online]. Available: http://www.eet.com/news/latest/showArticle.jhtml?articleID=55300884

[2] V. del Barrio, C. Gonzalez, J. Roca, A. Fernandez, and E. ATTILA, "a cycle-level execution-driven simulator for modern GPU architectures," in *Intl Symp. on Performance Analysis of Systems and Software*, 2006, pp. 231–241.

[3] D. Crisu, S. Cotofana, S. Vassiliadis, and P. Liuha, "Efficient hardware for tile-based rasterization," in *Proceedings of the 15th Annual Workshop on Circuits, Systems, and Signal Processing, Veldhoven*. Citeseer, 2004, pp. 352–357.

[4] ——, "Design Tradeoffs for an Embedded OpenGL-Compliant Hardware Rasterizer," in *Proc. of the 14 th Annual Workshop on Circuits, Systems, and Signal Processing. Veldhoven, The Netherlands:[s. n.]*. Citeseer, 2003.

[5] T. Akenine-Möller, E. Haines, and N. Hoffman, *Real-Time Rendering 3rd Edition*. Natick, MA, USA: A. K. Peters, Ltd., 2008.

[6] M. Segal and K. Akeley, "The opengl graphics system: A specification," 2009, version 3.2 (Core Profile).

[7] J. Pineda, "A parallel algorithm for polygon rasterization," *ACM SIGGRAPH Computer Graphics*, vol. 22, no. 4, p. 20, 1988.

[8] T. Akenine-Möller, "Mobile graphics hardware," 2007, draft for the course EDA075 Mobile Computer Graphics.

[9] M. McCool, C. Wales, and K. Moule, "Incremental and hierarchical Hilbert order edge equation polygon rasterization," in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*. ACM New York, NY, USA, 2001, pp. 65–72.

[10] K. Akeley, "Reality engine graphics," in *Proceedings of the 20th annual conference on Computer graphics and interactive techniques*. ACM, 1993, p. 116.

[11] D. B. Thomas and W. Luk, *Implementing Graphics Shaders Using FPGAs*. Lecture Notes in Computer Science. Springer Berlin, 2004.

[12] J. Roca, V. Moya, C. Gonzalez, V. Escandell, A. Murciego, A. Fernandez, and R. Espasa, "A SIMD-efficient 14 instruction shader program for high-throughput microtriangle rasterization," *The Visual Computer*, pp. 1–13.