Scene Management Models and Overlap Tests for Tile-Based Rendering

I. Antochi, B. Juurlink, S. Vassiliadis
Computer Engineering Laboratory
Delft University of Technology
Mekelweg 4, 2628 CD Delft, The Netherlands
E-mail: {tkg, benj, stamatis}@ce.et.tudelft.nl

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

Abstract

Tile-based rendering (also called chunk rendering or bucket rendering) is a promising technique for low-power, 3D graphics platforms. This technique decomposes a scene into smaller regions called tiles and renders the tiles one-by-one. The advantage of this scheme is that a small memory integrated on the graphics accelerator can be used to store the color components and z values of one tile, so that accesses to the frame and z buffer are local, on-chip accesses which consume significantly less power than off-chip accesses. Tile-based rendering, however, requires that the primitives (commonly triangles) are sorted into bins corresponding to the tiles. This paper describes several algorithms for sorting the primitives into bins and evaluates their computational complexity and memory requirements. In addition, we present and evaluate several tests for determining if a triangle and a tile overlap. Experimental results obtained using several suitable 3D graphics workloads show that various trade-offs can be made and that, usually, better performance can be obtained by trading it for memory. This information allows the designer to select the appropriate method depending on the amount of memory available and the computational power.

1. Introduction

A huge market is foreseen for wireless, interactive 3D graphics applications, in particular games[3]. Because contemporary wireless devices do not have sufficient computational power to support 3D graphics in real time and because present accelerators consume too much power, several companies and universities started to develop low-power 3D graphics accelerators[14, 9]. Tile-based rendering architectures appear to be promising for low-power implementation, because they employ a small, on-board memory to render a scene instead of a large, off-chip frame buffer. Tile-based accelerators, however, require a large *scene buffer* to store the primitives to be rendered. Furthermore, the primitives have to be sent to the accelerator on a per tile basis. In other words, they have to be sorted into bins that correspond to the tiles.

In this paper we present several algorithms to manage the scene buffer and evaluate their time and memory complexity. Sorting and sending the primitives involves two steps. First, the primitives are generated and buffered. After that, the primitives are sent to the graphics accelerator in tile-based order. By performing different computations in each step, various implementations with different time and memory complexity are possible. For example, one approach is to sort the primitives while they are buffered. This approach, however, requires a substantial amount of additional memory because primitives generally cover several tiles. Another approach is to leave the primitives unsorted but to sort them while they are sent to the accelerator. This approach requires no additional memory but generally consumes more time than the previous algorithm. We show that several intermediate approaches are possible as well.

In addition, we present and evaluate several tests that determine if a primitive (in particular triangle) and a tile overlap. Some of these tests have been proposed before in a different context but we adapted them for testing if a triangle and a tile overlap. Although some of these tests are computationally more expensive than the commonly employed bounding box test, they provide more accurate information, which implies that less memory is needed and fewer triangles need to be rendered by the graphics accelerator.

This paper is organized as follows. Section 2 briefly describes related work. In Section 3 we describe the triangle to tile overlap tests we considered for our implementation. The algorithms for scene management are presented in Section 4. Experimental results are presented in Section 5, and conclusions are given in Section 6.

Proceedings of the EUROMICRO Systems on Digital System Design (DSD'04)
0-7695-2203-3/04 $ 20.00 IEEE

IEEE COMPUTER SOCIETY

2. Related Work

Tile-based architectures were initially proposed for high-performance, parallel renderers [8, 11, 10]. Since tiles cover non-overlapping parts of a scene, the triangles that intersect with these tiles can be rendered in parallel. In such architectures it is important to balance the load on the parallel renderers [12]. These studies are, however, not very related to our study since we consider a low-power architecture in which the tiles are rendered sequentially one-by-one.

Tile-based rendering has also been used in power-aware architectures [14, 9]. However, no details have been provided on how the primitives are sorted into bins corresponding to the tiles. Furthermore, the only triangle-to-tile overlap test we could find in literature is the bounding box test [14, 9, 5, 4]. However, algorithms developed for other purposes (e.g. antialiasing [15] or collision detection [7, 1]) can be adapted for efficient primitive to tile sorting.

3. Overlap Tests

In this section we describe several triangle to tile overlap tests. Our contribution consist of introducing an overlapping test based on [15], but with the difference that our test does not require any coverage mask.

We remark that some of the described tests perform only partial classifications. They may yield a positive result even if a triangle does not intersect a tile. This conservative approach is preferable to predicting that a triangle and a tile do not overlap while in reality they do, because then "holes" may appear in the rendered scene if triangles have been discarded incorrectly. As can be expected, in general, more accurate tests are computationally more expensive. This allows various algorithms with different time and memory complexities to be developed (Section 4).

3.1. Bounding Box Test (BBOX)

This test determines if the axis aligned bounding box of a triangle intersects with the tile. This is illustrated in Figure 1. If the bounding box of the triangle does not overlap with the tile then also the triangle does not overlap with the tile. However, if the bounding box of the triangle overlaps with the tile, the triangle might overlap the tile, but generally there is no precise answer so in this case additional tests are required or it can be assumed that the triangle overlaps the tile and it can be sent to the rasterizer (in this case the rasterization of the triangle might generate no fragments if the triangle does not overlap with the tile). If a triangle is small, the BBOX test can be accurate (actually the accuracy of this test depends also on the thinness and orientation of a triangle) while for larger triangles the accuracy might drop

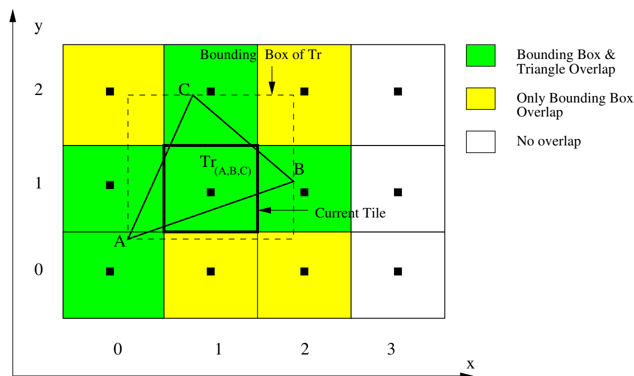

Figure 1. Triangle to tile BBOX test

significantly. We experimentally determined that, on common workloads, the BBOX test can generate up to 30% false intersections for large triangles.

3.2. Linear Edge Function Test (LET)

This test employs *edge functions* [13] to determine if a triangle intersects a tile. Edge functions are normally used to determine if a point is inside a triangle or, for instance, to compute a coverage mask for antialiasing[15]. In our case, we extended the equations presented in [15] so that no coverage mask is needed to determine if a triangle intersects a tile.

Consider a 2D vector defined by two points $A(X, Y)$ and $B(X + dX, Y + dY)$, and a line L_{AB} that passes through the two points. The edge function for a certain point (x, y) is defined as:

$$E_{L_{AB}}(x, y) = (x - X) \cdot dY - (y - Y) \cdot dX. \quad (1)$$

The edge function can be also be written using an incremental form as:

$$E_{L_{AB}}(x + \delta x, y + \delta y) = E_{L_{AB}}(x, y) + \delta x \cdot dY - \delta y \cdot dX. \quad (2)$$

The incremental form can be used to evaluate the edge function for a sequence of points more efficiently. In this case, only for the first point the edge needs to be computed using Equation (1) while for the rest of the points the incremental form can be used. The incremental computation of the edge function requires fewer operations than Equation (1). Furthermore, commonly $\delta x = 1$ or $\delta y = 1$, in which case, the incremental version requires only one multiplication.

The edge function can be used to determine the position of a point (x, y) relative to the line L_{AB} as follows:

IEEE COMPUTER SOCIETY

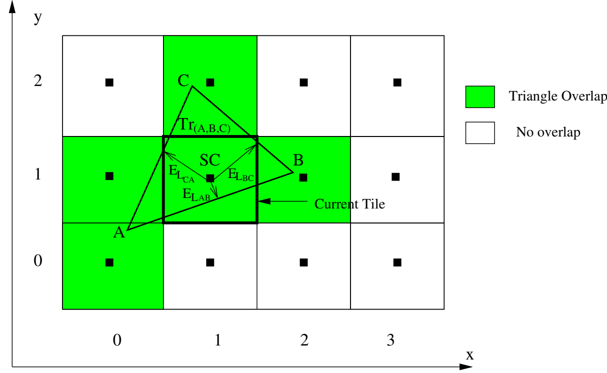

Figure 2. Triangle to tile test using linear functions

$$
\begin{array}{ll}
\text{If } E_{L_{AB}}(x,y) > 0 & \text{then the point is to the right of } L_{AB} \\
\text{If } E_{L_{AB}}(x,y) = 0 & \text{then the point is on } L_{AB} \\
\text{If } E_{L_{AB}}(x,y) < 0 & \text{then the point is to the left of } L_{AB}
\end{array}
\tag{3}
$$

The conditions that can be used (but are not sufficient) to determine if a counter-clockwise oriented triangle T, defined by three vertices $A(x_A, y_A)$, $B(x_B, y_B)$, $C(x_C, y_C)$, intersects a square S defined by a center point $CS(x_{cs}, y_{cs})$ and having a total width of l are:

$$
\begin{array}{l}
E_{L_{AB}}(x_{cs}, y_{cs}) \leq \frac{l}{2} \cdot (|x_B - x_A| + |y_B - y_A|) \\
E_{L_{BC}}(x_{cs}, y_{cs}) \leq \frac{l}{2} \cdot (|x_C - x_B| + |y_C - y_B|) \\
E_{L_{CA}}(x_{cs}, y_{cs}) \leq \frac{l}{2} \cdot (|x_A - x_C| + |y_A - y_C|)
\end{array}
\tag{4}
$$

Considering that a tile can be regarded as a rectangle R defined by a center point $M(x_m, y_m)$, and a width of w and a height of h, we can transform the rectangle R into a square S with a width of 1 using a normalization (division by (w, h)). By performing the same operation on the vertices of the triangle T, the conditions presented in Equations (4), where $l = 1$ can be used to test if the triangle intersects the tile. Because we do not use any masking mechanism as used in [15], the Equations (4) are not sufficient to correctly classify the overlap of a triangle with a tile. In order to eliminate false intersections, a bounding box test should be performed first and followed by the Equations (4). Figure 2 depicts the outcome of the triangle to tile test using linear functions. The triangle to tile test using LET test is exact as opposed to the BBOX test where triangles might have been classified as overlapping a tile even if there was no real triangle to tile overlap.

We remark that there are more tests to accurately classify the triangle to tile overlaps. For example, the Separating Axes Test (SAT) test described in [1, 7], and used in the implementation of the Chromium [10] parallel rendering system. The SAT test is based on the observation that two *convex* objects do not overlap if there exists a line for which their respective projections on the line do not intersect. It is shown in [6] that is enough to consider the normals to the edges of each polygon as projection lines. This method, however, even after aggressive optimizations remained more computationally intensive than the LET method and it was not further employed in our study.

4. Scene Management Algorithms

In this section we present several algorithms for sorting primitives into bins corresponding to tiles and determine their computational complexity and memory requirements. As described in Section 1, sorting the primitives into bins and sending them to the rasterizer involves two stages. In the first stage the primitives are buffered and an initial sorting step may be performed. In the second stage the primitives are sent to the rasterizer in tile-based order, eventually after a second sorting step.

Algorithm DIRECT This algorithm simply scans the whole list of primitives for each tile and sends the primitives that (potentially) overlap the current tile to the rasterizer.

for each triangle *Tr*
    buffer *Tr*

for each tile T
    for each triangle *Tr*
        compute bbox of *Tr*
        if bbox of *Tr* and T overlap
            send *Tr*

Let $\#tiles$ and $\#triangles$ be the number tiles and triangles, respectively. The time complexity of algorithm DIRECT is

$$
\begin{array}{l}
t_{buf} \cdot \#triangles + \\
t_{bbox\text{-}total} \cdot \#tiles \cdot \#triangles + \\
t_{send} \cdot \#triangles \cdot bbox_overlap,
\end{array}
$$

where t_{buf} is the cost of placing a triangle in the scene buffer, $t_{bbox\text{-}total}$ is the cost of computing the bounding box of a triangle and determining if a bounding box and a tile overlap, t_{send} is the cost of sending a triangle to the rasterizer, and *bbox_overlap* is the overlap factor (i.e., the average number of tiles a triangle covers) if the bounding box test is employed.

The main advantage of algorithm DIRECT is that it requires no memory in addition to the scene buffer. We also remark that here we used the bounding box test, but other tests may also be employed.

IEEE COMPUTER SOCIETY

Algorithm TWO_STEP In this algorithm the bounding box of each triangle is computed and stored during the buffering stage. This avoids having to recompute the bounding box for each triangle/tile tuple during the sending stage. It requires, however, that the bounding box of each primitive is kept.

```
for each triangle Tr
    buffer Tr
    compute and store bbox of Tr

for each tile T
    for each triangle Tr
        if bbox of Tr and T overlap
            send Tr
```

The complexity of this algorithm is

$$(t_{buf} + t_{bbox\text{-}compute}) \cdot \#triangles + \\ t_{bbox\text{-}test} \cdot \#tiles \cdot \#triangles + \\ t_{send} \cdot \#triangles \cdot bbox_overlap,$$

where $t_{\text{bbox-compute}}$ is the cost of computing the bounding box of a triangle and $t_{\text{bbox-test}}$ is the cost of testing if a bounding box of a triangle and a tile overlap (so $t_{bbox\text{-}total} = t_{bbox\text{-}compute} + t_{bbox\text{-}test}$). We assumed that the cost of storing a bounding box is negligible since a storing operation is performed by default while computing the bounding box components. However, even if a separate bounding box storing cost is added, its contribution to the total cost (complexity) is negligible.

The amount of additional memory required by algorithm TWO_STEP is $\#triangles \cdot sizeof(bbox)$, where $sizeof(bbox)$ is the size of a bounding box structure (i.e. 4 integers).

Algorithm TWO_STEP_LET This algorithm is similar to the TWO_STEP algorithm described above. The difference is that in the second stage a LET overlap test instead of just BBOX is used. Since the LET test contains a BBOX test, the main LET test (Equations 4) is applied only to triangles that have passed the BBOX test.

```
for each triangle Tr
    buffer Tr
    compute and store bbox of Tr

for each tile T
    for each triangle Tr
        if LET test indicates Tr and T overlap
            send Tr
```

The complexity of this algorithm is

$$(t_{buf} + t_{bbox\text{-}compute}) \cdot \#triangles + \\ t_{bbox\text{-}test} \cdot \#tiles \cdot \#triangles + \\ t_{let\text{-}test} \cdot \#triangles \cdot bbox_overlap + \\ t_{send} \cdot \#triangles \cdot let_overlap,$$

where $t_{let\text{-}test}$ is the cost of testing if a triangle and a tile overlap using LET test, and *let_overlap* is the LET overlap factor (i.e., the average number of tiles covered by a triangle if the LET test is employed).

While the TWO_STEP_LET algorithm takes more time than the TWO_STEP algorithm, the number of triangles sent to the rasterizer by the TWO_STEP_LET algorithm ($\#triangles \cdot let_overlap$) is lower than or equal to the number of triangles sent to the rasterizer in the TWO_STEP algorithm ($\#triangles \cdot bbox_overlap$) since the LET test is accurate while the BBOX test is approximative. By sending less triangles to the accelerator this algorithm reduces the computational requirements at the accelerator.

The amount of additional memory required by algorithm TWO_STEP_LET is the same as for the TWO_STEP algorithm.

Algorithm SORT In this algorithm for each tile there is a buffer with pointers to the primitives that overlap the tile according to the BBOX test. For each tile only the primitives that have a pointer in the corresponding tile buffer will be sent to the rasterizer.

```
for each triangle Tr
    buffer Tr
    compute bbox of Tr
    for each tile T that overlaps bbox of Tr
        insert pointer to Tr in the buffer of T

for each tile T
    for each triangle Tr in the buffer of T
        send Tr
```

The complexity of this algorithm is

$$(t_{buf} + t_{bbox\text{-}compute}) \cdot \#triangles + \\ t_{insert} \cdot \#triangles \cdot bbox_overlap + \\ t_{tiletrav} \cdot \#tiles + \\ t_{send} \cdot \#triangles \cdot bbox_overlap,$$

where t_{insert} is the cost of inserting a pointer to a triangle in the buffer of the tile. There is no need to add a $t_{bbox\text{-}test}$ cost since there is no BBOX test performed (we can determine from the bounding box coordinates in which tiles to insert pointers). The $t_{tiletrav}$ is the cost to traverse a tile.

The amount of additional memory required by the SORT algorithm is $\#triangles \cdot bbox_overlap \cdot 2 \cdot sizeof(pointer) +$

IEEE COMPUTER SOCIETY

$\#tiles \cdot 2 \cdot sizeof(pointer)$, where *sizeof(pointer)* denotes the size of a pointer (4 bytes). In our current implementation we use a (prealocated) linked list of pointers to primitives, thus we need to store two pointers for each primitive (one pointing to the primitive and one to the next primitive in the tile). We also use two pointers for each tile (to the first primitive for the tile and the last primitive inserted).

Algorithm SORT_LET This algorithm is similar to the previous algorithm. The difference is that eventually the LET test is used to determine if a triangle overlaps a tile. For each tile only the primitives that have a pointer in the corresponding tile buffer will be sent to the rasterizer.

for each triangle *Tr*
  buffer *Tr*
  compute bbox of *Tr*
  for each tile *T* that overlaps bbox of *Tr*
    if LET test indicates *Tr* and T overlap
    insert pointer to *Tr* in the buffer of *T*

for each tile *T*
  for each triangle *Tr* in the buffer of *T*
    send *Tr*.

The complexity of this algorithm is

$$\begin{aligned}&(t_{buf} + t_{bbox\text{-}compute}) \cdot \#triangles + \\ &t_{let\text{-}test} \cdot \#triangles \cdot bbox_overlap + \\ &t_{insert} \cdot \#triangles \cdot let_overlap + \\ &t_{tiletrav} \cdot \#tiles + \\ &t_{send} \cdot \#triangles \cdot let_overlap,\end{aligned}$$

The amount of additional memory required by the SORT_LET algorithm is $\#triangles \cdot let_overlap \cdot 2 \cdot sizeof(pointer) + \#tiles \cdot 2 \cdot sizeof(pointer)$.

5. Experimental Results

In order to determine the computational and memory requirements of the algorithms, we have simulated several traces of 3D graphics applications and measured certain statistics such as the BBOX and LET overlap factors. Furthermore, we estimated most parameters such as t_{buf} and t_{insert} by counting the number of elementary operations (assignments, comparisons, etc.) required to implement the operation. Other parameters such as $t_{\text{bbox_test}}$ can vary because the implementation of this operation contains if-then-else statements which implies that the time depends on the control flow. In order to estimate these parameters, we wrote a program that performs these tests and inserted counters to determine how often each branch was executed. These statistics have been subsequently substituted in the complexity formulae of the algorithms. We remark that cycle-accurate simulations of the algorithms on all workloads are not feasible, because that is too time consuming.

This section is organized as follows. Section 5.1 describes the benchmarking suite. The efficiency of the overlap tests is discussed in Section 5.2. Section 5.3 presents the values of the statistics and the parameters used to calculate the time and memory required by each algorithm. Finally, the runtime results and memory requirements of the algorithms are presented and discussed in Section 5.4.

5.1. Benchmarking Suite

In order to compare the efficiency of the proposed algorithms we used the benchmarking suite proposed in [2]. It consists of 7 components: Q3L, Q3H, Tux, Aw, ANL, GRA, and DIN. The Q3L profile corresponds to a low resolution (320x240) demo of the Quake III 3D FPS game. The Q3H profile is based on the same demo as Q3L only that it uses higher resolution (640x480). Tux is a 3D racing game (guide a penguin) available on Linux platforms. The Aw (Awadvs-04) profile is part of the Viewperf 6.1.2 package. The ANL, GRA, and DIN are 3D VRML models for which "fly-by" scenes were created and traced.

The traces were fed to our modified Mesa library. The Mesa library performed primitive backface culling and generated lists of remaining primitives. The list of primitives were sent to our tile-based accelerator simulator, where different primitive to tile algorithms were used. We used a tile size of 32x16 pixels (as used in current tile-based hardware accelerators [14]), and the window sizes were 320x240 for Q3L, and 640x480 for the other benchmark suite components.

5.2. Efficiency of the Overlap Tests

For each frame, all tiles are rendered sequentially and only when all the tiles have been rendered, a new frame is processed. We used two primitive to tile repartition strategies. The BBOX strategy corresponds to a simple bounding box primitive to tile overlap test. This strategy, as mentioned in Section 3.1 might not produce the optimal (lowest) number of primitives to be sent to the rasterizer, but the computational overhead of this test is generally low. The LET strategy corresponds to a linear edge test for primitive to tile overlap. The LET strategy produces an accurate number of primitives that should be sent to the tile-based rasterizer.

Table 1 presents the number of I/O write accesses to the accelerator based on our simulator using the BBOX test or LET test. The total I/O numbers represent the number of accesses due to state-changing information (e.g. enable or disable depth test) plus the number of I/O due to sending

triangles. Since the number of primitives sent to the accelerator depends only on the overlapping test performed, these numbers are independent of the algorithm used. The number of I/O writes needed to sent triangles to the accelerator for the BBOX test was up to 62% larger than for the LET test. Another interesting result from this table is that the LET test not only reduces the number of I/Os, but also the number of state changes. This happens due to the fact that we used a *lazy* update mechanism for the state change (the state changes are committed to the renderer when they are needed, thus if some of the primitives are discarded some state changes might not be required). Thus, using a more accurate overlapping test is beneficial also for the amount of state change information sent to the accelerator.

5.3. Parameters and Benchmark Statistics

Some of the parameters used to estimate the complexity of the algorithms ($t_{bbox_compute}, t_{bbox_test}, t_{let_test}$) can vary across the workloads. In order to reduce the errors obtained by estimating them statistically, we wrote programs to compute the average number of elementary operations needed to implement each test and obtained particular values for each workload. The results are presented in Table 2. It can be seen that the obtained $t_{bbox_compute}$ and t_{bbox_test} parameters are quite uniform across the workloads while t_{let_test} has a larger variation.

Other parameters of the workloads such as the average or maximum number of triangles per frame are presented in Table 3. The *average triangles / frame* statistics represents the average number of triangles sent from the Mesa library to our driver after backface culling. This number is actually the $\#triangles$ parameter used to compute the complexity of the algorithms. The *max. triangles* represent the maximum number of triangles sent for one frame. This number can be used to determine the maximum amount of memory required to buffer the triangles for one frame. This number can also be used to determine the computational power required for real-time operation.

For the other parameters, the following assumptions were employed: $t_{buf} = 50$, $t_{insert} = 6$ (two additions, three assignments, and one comparison), $t_{tiletrav} = 4$ (two comparisons, one assignment, and one increment), $t_{send} = 40$ (the number of I/O writes currently used to transfer the data for a triangle in our simulator).

5.4. Runtime Results and Memory Requirements

This section presents the results we obtained for triangle to tile repartition algorithms. We were not able to perform a cycle accurate simulation of the workload due to the fact that it is too time consuming.

The average time taken by each scene management algorithm to process one frame of every benchmark is presented in Table 4, while Figure 3 depicts the time required by each algorithm relative to the amount of time taken by algorithm DIRECT. As expected, algorithm DIRECT requires the largest number of operations by far, while SORT takes the least amount of time. On average, across all benchmarks, the SORT algorithm is 44 times as fast as DIRECT. The TWO_STEP algorithm, even though it also scans the entire scene buffer for each tile, has reasonable performance. It is slower than algorithm SORT by a factor of 6 on average. It can also be observed that algorithm TWO_STEP_LET is hardly slower than TWO_STEP and, therefore, preferable, since it sends fewer triangles to the rasterizer which means that the computational load on the rasterizer is reduced. Algorithm SORT_LET, on the other hand, is slower than algorithm SORT by a factor of 1.6 on average.

The amount of memory required by each algorithm, in addition to the scene buffer which is needed to buffer the primitives, is presented in Table 5. It is also visually depicted in Figure 4. As explained before, algorithm DIRECT does not require any additional memory. Furthermore, as expected, the SORT algorithm needs the most memory, since the amount of additional memory it requires is proportional to the number of triangles and the BBOX overlap factor (the average number of tiles covered by a triangle if the BBOX test is employed). Because the LET test is exact while the BBOX test is not, SORT_LET requires less memory than SORT. However, the difference is significant only for one benchmark (Q3H) for which SORT needs almost twice the amount of additional memory as SORT_LET. For the other benchmarks the difference is much smaller (a factor of 1.17 on average). The reason is that the BBOX test is rather exact for all benchmarks except Q3H. The TWO_STEP and TWO_STEP_LET algorithms require the same amount of memory and are, therefore, depicted together. On average, TWO_STEP requires a factor of 3.2 less additional memory than SORT. However, this difference depends strongly on the benchmark. For benchmarks with a small overlap factor (e.g., Aw), the difference is hardly significant, while for benchmarks with a large overlap factor (in particular Q3H) the difference is considerable.

Which algorithm is preferable depends, of course, on the computational power and the amount of memory of a particular implementation. DIRECT is probably not a practical algorithm because it has poor performance. Our results indicate that TWO_STEP_LET is better than TWO_STEP, since they require the same amount of memory and because TWO_STEP_LET takes only a bit more time than TWO_STEP. Furthermore, TWO_STEP_LET sends fewer triangles to the rasterizer, which implies that the rasterizer has to perform less work. SORT and SORT_LET take

IEEE COMPUTER SOCIETY

Table 1. I/O writes for various overlapping tests

	Q3L	Q3H	Tux	Aw	ANL	GRA	DIN
BBOX total I/O	701M	1,907M	664.5M	342M	524M	357M	215M
BBOX triangles I/O	432M	1,024M	320M	265M	338M	152M	160M
LET total I/O	583M	1,394M	565M	337M	459M	322M	206M
LET triangles I/O	340M	633M	235M	260M	281M	131M	150M

Table 2. Complexity parameters for each workload

	Q3L	Q3H	Tux	Aw	ANL	GRA	DIN
$t_{bbox_compute}$	14.02	14	13.98	14.25	14.25	14.25	14.20
t_{bbox_test}	2.08	1.995	1.780	1.86	1.93	1.77	1.78
t_{let_test}	50.5	43.40	48.93	59.15	52.9	54.66	57.79

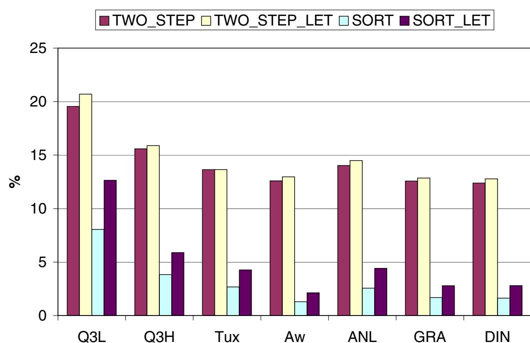

Figure 3. Time taken by each scene management algorithm, relative to the amount of time taken by algorithm DIRECT

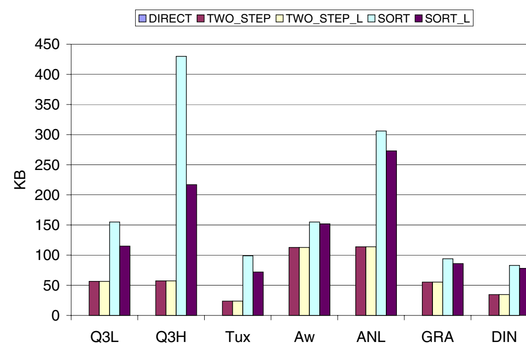

Figure 4. Memory requirements of the scene management algorithms

less time than TWO_STEP_LET, but require more memory. So the 3D graphics system designer has to make a trade-off between these algorithms. If SORT_LET is preferable to SORT or vice versa can also not be assured. Although SORT_LET sends fewer triangles to the rasterizer, SORT requires significantly less time than SORT_LET while SORT_LET reduces the memory requirements only marginally for all but one benchmark.

6. Conclusions

A two stage model for triangle to tile repartition for tile-based graphics accelerators has been presented. In addition, different algorithms to test the triangle to tile overlap have been described. By comparing factors like computational power or memory required, a 3D graphics accelerator designer can chose the most suitable algorithm for an implementation. While the number of primitives sent to the accelerator depends only on the triangle to tile overlap test used, the memory required to sort and store the primitives before being sent to the accelerator and also the computational power depends largely on the algorithm employed.

The DIRECT algorithm is probably not a practical algorithm because is has poor performance. However, the TWO_STEP algorithm, even though it also scans the entire scene buffer for each tile, has reasonable performance while it does not require a large amount of additional memory. If the computational performance is more important than additional memory required, then the SORT algorithm is more suitable. To determine the best algorithm for a particular implementation it should be taken into consideration also the fact that using the LET overlap test, instead of BBOX, more computational power is required at the scene management stage, but depending on the computational power required to process and discard additional triangles on the accelerator it might be a better choice to use LET instead of BBOX since the computations at the accelerator to discard the additional triangles generated by BBOX can be higher.

References

[1] T. Akenine-Möller and E. Haines. *Real-Time Rendering (second edition)*. A.K. Peters Ltd., 2002.

[2] I. Antochi, B. Juurlink, S. Vassiliadis, and P. Liuha. GraalBench: A 3D Graphics Benchmark Suite for Mobile Phones. In *Proc. ACM SIGPLAN/SIGBED Conf. on Languages,*

IEEE COMPUTER SOCIETY

Table 3. Relevant characteristics of the benchmarks

	Q3L	Q3H	Tux	Aw	ANL	GRA	DIN
number of frames	1,379	1,379	1,363	603	600	599	600
average triangles / frame	3,350	3,436	1,825	11,053	4,455	3,681	4,150
max. triangles / frame	7,074	7,170	2,980	14,102	14,236	6,907	4,313
max. bbox / frame	19,288	53,154	11,789	18,822	37,771	11,233	9,858
max. let / frame	14,175	26,542	8,478	18,465	33,469	10,109	9,088
bbox_overlap	3.06	7.03	4.17	1.34	4.08	2.28	2.06
let_overlap	2.39	4.32	3.05	1.32	3.4	1.98	1.95
max. scene buffer memory required	594k	602k	250k	1,185k	1,196k	580k	362k

Table 4. Number of elementary operations per frame for each scene management algorithm.

	Q3L	Q3H	Tux	Aw	ANL	GRA	DIN
DIRECT	8.7M	34M	17.6M	108M	44.2M	35.8M	40.7M
TWO_STEP	1.7M	5.3M	2.4M	13.6M	6.2M	4.5M	5.0M
TWO_STEP_LET	1.8M	5.4M	2.4M	14M	6.4M	4.6M	5.2M
SORT	0.7M	1.3M	0.47M	1.4M	1.13M	0.6M	0.66M
SORT_LET	1.1M	2.0M	0.75M	2.3M	1.9M	1.0M	1.1M

Table 5. Additional maximum memory requirements (bytes) per frame for each scene management algorithm.

	Q3L	Q3H	Tux	Aw	ANL	GRA	DIN
DIRECT	0	0	0	0	0	0	0
TWO_STEP/ TWO_STEP_LET	56.6k	57.4k	23.8k	112.8k	113.9k	55.26k	34.5k
SORT	155k	430k	99k	155k	306k	94k	83k
SORT_LET	115k	217k	72k	152k	273k	86k	78k

Compilers, and Tools for Embedded Systems (LCTES'04), pages 1–9. ACM Press, June 2004.

[3] ARM Ltd. ARM 3D Graphics Solutions. Available at http://www.arm.com.

[4] M. Chen, G. Stoll, H. Igehy, K. Proudfoot, and P. Hanrahan. Simple Models of the Impact of Overlap in Bucket Rendering. In *Proc. ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware*, pages 105–112, Lisbon, Portugal, 1998. ACM Press.

[5] M. Cox and N. Bhandari. Architectural Implications of Hardware-Accelerated Bucket Rendering on the PC. In *Proc. 1997 SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 25–34. ACM Press, 1997.

[6] D. H. Eberly. Intersection of Convex Objects: The Method of Separating Axes. http://www.magic-software.com/.

[7] D. H. Eberly. *3D Game Engine Design: A Practical Approach to Real-Time Computer Graphics*. Morgan Kaufmann/Academic Press, 2001.

[8] H. Fuchs, J. Poulton, J. Eyles, T. Greer, J. Goldfeather, D. Ellsworth, S. Molnar, B. T. G. Turk, and L. Israel. Pixel-Planes 5: A Heterogeneous Multiprocessor Graphics System Using Processor-Enhanced Memories. *Computer Graphics, Vol. 23, No. 3, pages. 79–88*, July 1989.

[9] E. Hsieh, V. Pentkovski, and T. Piazza. ZR: A 3D API Transparent Technology for Chunk Rendering. In *Proc. 34th ACM/IEEE Int. Symp. on Microarchitecture MICRO-34*, 2001.

[10] G. Humphreys, M. Houston, R. Ng, R. Frank, S. Ahern, P. D. Kirchner, and J. T. Klosowski. Chromium: A Stream Processing Framework for Interactive Rendering on Clusters. In *Proc. 29th Annual Conf. on Computer Graphics and Interactive Techniques (SIGGRAPH 2002)*, pages 693–702, 2002.

[11] S. Molnar, M. Cox, D. Ellsworth, and H. Fuchs. A Sorting Classification of Parallel Rendering. *IEEE Comput. Graph. Appl.*, 14(4):23–32, 1994. IEEE Computer Society Press.

[12] C. Mueller. The Sort-First Rendering Architecture for High-Performance Graphics. In *Proc. 1995 Symp. on Interactive 3D Graphics*, pages 75–84. ACM Press, 1995.

[13] J. Pineda. A Parallel Algorithm for Polygon Rasterization. In *Proc. 15th Annual Conference on Computer Graphics and Interactive Techniques*, pages 17–20. ACM Press, 1988.

[14] PowerVR. 3D Graphical Processing (Tile Based Rendering - The Future of 3D), White Paper. http://www.beyond3d.com/reviews/videologic/vivid/PowerVR_WhitePaper.pdf, 2000.

[15] A. Schilling. A New Simple and Efficient Antialiasing With Subpixel Masks. In *Proc. 18th Annual Conference on Computer Graphics and Interactive Techniques*, pages 133–141. ACM Press, 1991.