Improving the Energy Efficiency of the Graphics Pipeline by Reducing Overshading

David Corbalán-Navarro[1], Juan L. Aragón[1], Martí Anglada[2], Enrique de Lucas[3], Joan-Manuel Parcerisa[2] and Antonio González[2]

Resumen— **The most common task of GPUs is to render images in real time. When rendering a 3D scene, a key step is determining which parts of every object are visible in the final image. There are different approaches to solve the visibility problem, the Z-Test being the most common in modern GPUs. A main factor that significantly penalizes the energy efficiency of a GPU, especially in the mobile arena, is the so-called *overshading*, which happens when a portion of an object is shaded and rendered but finally occluded by another object. This useless work results in a waste of energy, however, the conventional Z-Test only eliminates a fraction of it.**

In this paper we present a novel microarchitectural technique, the Ω-Test, to drastically reduce overshading on a Tile-Based Rendering (TBR) architecture. The proposed approach leverages frame-to-frame coherence by taking advantage of the costly and valuable calculations made in previous frames. In particular, we propose to reuse information from the Z-Buffer of the previous frame, which is currently discarded. We make the observation that due to the existing frame-to-frame coherence, the Z-Buffer of a frame will have a high similarity in many areas with that of the previous frame. As a result, the proposed technique avoids many costly computations and off-chip memory accesses. Our experimental evaluation shows that Ω-Test reduces the average energy consumption of the overall GPU/Memory system by 15.7 % and the runtime of the evaluated benchmarks by 10.6 % on average.

Palabras clave— **Graphics processors, Mobile processors, Portable devices, Hardware architecture, Processor architecture, Energy-aware systems, Low-power design, Hidden line/surface removal, Visibility determination.**

I. Introduction

Mobile devices, such as smartphones, tablets or smartwatches, have undergone a major evolution over the recent years. Users increasingly demand more complex applications on such devices, which requires higher performance designs at the expense of negatively impacting their autonomy. As a consequence, the energy efficiency is one of the most important aspects in mobile devices [1], [2], especially for graphics applications such as modern 3D games, for which visual quality, richer graphics details, higher screen resolutions, and smooth movements are crucial for the best user experience.

One of the most energy-consuming components on current SoCs is the GPU (Graphics Processing Unit) [3], [4]. To provide a better insight, Figure 1 shows the power breakdown for a conventional Tile-Based Rendering (TBR) architecture. In particular, both the accesses to main memory and the activity of the Fragment Processors are by far the two major contributors, responsible for 53 % and 42 % respectively of the overall GPU power whereas the Vertex Processors incur a very minor energy consumption (2 %) [5]. Nevertheless, the fragments processed in a scene by the Fragment Processors outnumber the amount of primitives by two orders of magnitude (our experimental results show a ratio of 125:1 for the evaluated benchmarks).

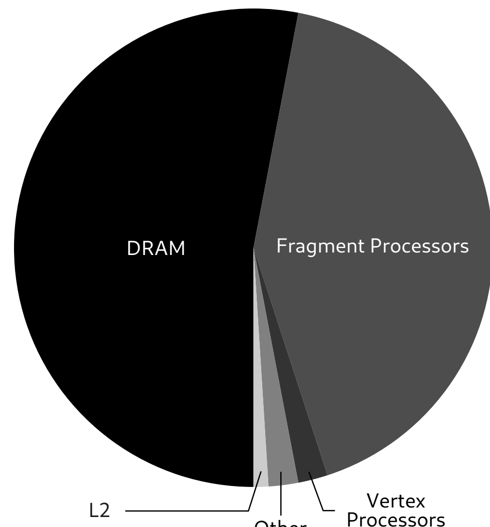

Fig. 1: Power breakdown for the baseline GPU employed in our experiments.

Given the huge number of fragments to be processed in every frame and the high computational cost of rendering one single fragment, it is crucial not to waste precious resources on shading fragments that will be later occluded by other primitives. For that reason, *visibility determination* is a fundamental task of the graphics pipeline in order to detect visible and occluded surfaces [6]. In particular, fragments that appear behind others, for a given camera viewpoint, are not visible in the final scene. The solution to the visibility problem is not unique and multiple approaches can be found in the literature [7], [8], [9] being the so-called Depth Test (also known as Z-Test) [10], which performs a visibility test at pixel granularity, the most widely implemented technique in contemporary GPUs.

While using the Depth Test ensures the correct visibility determination regardless of the order in which the scene is processed, it does not guarantee that

[1]Dpto. de Ingeniería y Tecnología de Computadores, Universidad de Murcia, e-mail: {dcorbalan, jlaragon}@ditec.um.es
[2]Departament d'Arquitectura de Computadors, Universitat Politècnica de Catalunya, e-mail: {manglada, jmanel, antonio}@ac.upc.edu
[3]Esperanto Technologies, Mountain View, CA, US, e-mail: enrique.delucas@imgtec.com

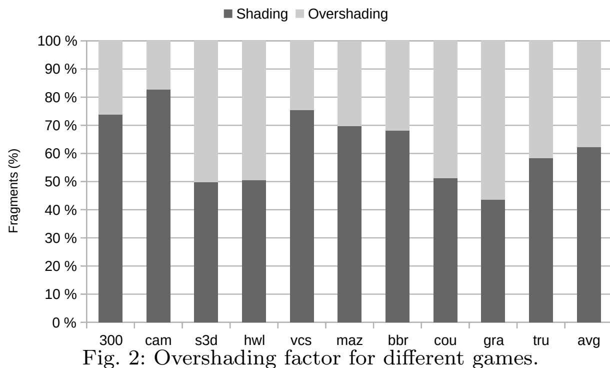

Fig. 2: Overshading factor for different games.

each pixel in the final screen is not rendered multiple times. This problem is commonly referred to as *overshading*, which is very common in games with complex scenes or poorly optimized. Overshading occurs when *more* than one opaque fragment is drawn in the same position on the screen, being only visible the closest to the camera viewpoint. Overshading is undesirable as it represents useless activity, and an early and accurate visibility determination can significantly improve performance and reduce the energy consumption. Ideally, the maximum potential that can be reached is to draw a single fragment per pixel (or screen position).

However, mobile GPUs (that commonly implement a TBR architecture [11]) heavily suffer from overshading since once the geometry stage is complete, all the primitives in a tile are rasterized and the fragments, before being rendered, perform an *early depth test* on-the-fly. In a worst-case scenario, in which primitives arrive in a back-to-front order, the Early Depth Test stage cannot avoid the rendering of any occluded fragments.

To provide an insight of the magnitude of this problem, Figure 2 shows the amount of overshading in a TBR architecture for a set of modern games (Section V will detail the evaluation methodology). In this work we define overshading as the fraction of fragments that are finally occluded over the total amount of processed fragments. It can be observed that 35 % of the shaded fragments on average are eventually occluded, with some games such as Counter Strike, Gravity, Sniper3D and Hot Wheels reaching an overshading factor around or over 50 %.

In this paper we propose the Ω-Test[1], a novel micro-architectural technique that attacks overshading and drastically reduces the amount useless work performed by the Fragment Processors. Our approach relies on exploiting the frame-to-frame coherence [12], [13] by reusing information from the Z values of the previous frame. Because of the small differences from frame to frame, we use information from the previous one to speculatively detect which fragments are occluded, instead of using only information from the current frame's Z-Buffer. Note, however, that our technique does not introduce any error in the final rendered image since fragments that are wrongly identified as occluded are later detected and rasterized.

Summarizing, the main contribution of this work is proposing a mechanism aimed at effectively reducing the overshading factor within a scene, decreasing the number of fragments processed as well as the costly memory accesses to textures that they would require. Hence, improving the energy efficiency of the GPU while decreasing the execution time. Our experimental results, for a commercial set of benchmarks, show the Ω-Test reduces the overall energy consumption of the GPU/Memory system by 15.7 % and achieves an average speedup of 1.106x.

The rest of the paper is organized as follows. Section II provides some background on the graphics pipeline of mobile GPUs and how the visibility problem is commonly solved. Sections III and IV describe the proposed Ω-Test approach and its implementation details. Section V describes our evaluation methodology while Section VI quantifies and analyzes the achieved performance and energy efficiency. Section VII reviews some relevant literature and, finally, Section VIII summarizes the main conclusions of the work.

II. Background

A. Tile-Based Rendering Architectures

The architecture of modern GPUs can be categorized into two main classes depending on how they process a scene: Immediate Mode Rendering (IMR) and Tile Based Rendering (TBR). While IMR processes and renders all of the primitives of a scene at once and it is the common design choice for high-end GPUs, TBR is aimed at improving the energy efficiency, and thus, it is commonly implemented in mobile GPUs. The key feature of TBR is that the screen area is divided in small regions of fixed size called *tiles*. This partitioning allows the tiles to be individually rendered and benefits from the use of much smaller on-chip buffers for storing depths and output color values, which dramatically reduces the amount of accesses to main memory and the energy consumption of the system.

Since our proposal targets mobile GPUs, TBR is the baseline architecture we have chosen for this work. Figure 3 shows the graphics pipeline of a TBR architecture, which is composed of two fundamental stages: the Geometry Pipeline and the Raster Pipeline. Both stages are serialized since there are strong dependencies between them, and the Tiling Engine acts as a mediator in between.

The Geometry Pipeline starts with a memory access stage to fetch the vertices of the scene. These vertices are transformed by geometric operations and assembled into primitives, typically triangles, which undergo a clipping process: primitives that lay out of the visible part of the scene (i.e., the part that the camera captures in its volume of vision also known as *frustum view*) are removed and/or cut accordingly. Additional steps such as *backface culling* can also be

[1]Named this way making an analogy with the Greek alphabet, where Ω is the last letter, analogously as Z is in the Latin alphabet.

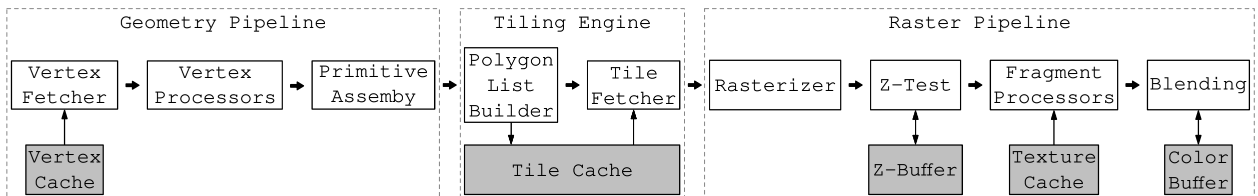

Fig. 3: Simplified version of the graphics pipeline for a TBR architecture.

applied to further reduce the number of primitives to be considered. Next, the Tiling Engine sorts primitives into tiles, i.e., each tile contains all the primitives that totally or partially fall inside the tile. These primitive lists are stored in memory and are the input data of the Raster Pipeline.

The Tiling Engine is in charge of scheduling the tiles to be processed by the Raster Pipeline (also referred to as Raster Unit). Note that multiple Raster Units can be used to process different tiles in parallel. The processing of a tile consists of several stages. First, the *rasterizer* tests the primitives at pixel granularity to determine the pixels covered by them. If a pixel is covered by a primitive, the *rasterizer* interpolates the value of the primitive's attributes at the pixel's position.

Fragments are grouped into quad fragments that are sent to the next stage of the pipeline, the Depth Test or Z-Test. The Z-Buffer stores the depths of all the fragments processed so far. To determine if the current fragment is occluded, its depth is compared with that stored in the same position of the Z-Buffer. The resulting visible quad fragments are sent to the Fragment Stage, which contains the Fragment Processors. A Fragment Processor executes a *shader* program which computes the color of each quad fragment. The resulting colors are stored in the Color Buffer. A Blending Unit allows for transparency effects by mixing the resulting colors with those already present in same Color Buffer position.

B. Visibility Determination

As cited earlier, the most common approach to determine the visibility is the Depth Test, performed at fragment granularity. Modern GPUs typically implement this test in a stage called *Early Depth Test* by using a Z-Buffer that stores a value for each position of the visible area. Each of these values is the depth of the nearest fragment of that position. Thus, when a fragment is going to be processed, this stage checks whether it is closer to the camera than the fragment already in the same position by comparing both depths. If the current fragment is further (deeper) than the existing one in the Z-Buffer, it is discarded, avoiding the costly shading and texturing process. Otherwise, the current fragment's depth is kept in the Z-Buffer (overwriting the previous depth value) meaning that the current fragment is the closest to the camera so far. Note the Z-Buffer for a tile is built on-the-fly, therefore, the Z-Test approach can effectively discard fragments when they arrive in a front-to-back order (i.e., later fragments that fall behind a closer one are discarded). However, it is totally ineffective when fragments arrive in a back-to-front order. In any case, the major advantage of the Depth Test is that it always leads to the correct final image regardless of the order in which fragments arrived to the Fragment Processors. The main drawback, on the other hand, is that it cannot avoid the high amount of overshading, as it was shown in Figure 2.

III. The Ω-Test Approach

The proposed Ω-Test slightly modifies the behavior of the Early Depth Test stage, where the visibility of fragments is determined. After performing the original Z-Test, and updating the Z-Buffer if necessary, a second test is performed but this time using an Ω-Buffer, a new structure similar to the Z-Buffer which holds the Z values corresponding to the previous frame. If the Ω-Test is passed, the fragment can proceed to shading. Otherwise, the fragment is discarded as we assume it will be occluded in the current frame, as it was indeed occluded in the previous frame (according to the contents of the Ω-Buffer which corresponds to, actually, the Z-Buffer of the previous frame).

Our technique reduces overshading with respect to TBR since each fragment has to pass a second test. This Ω-Test can be seen as a *backup* test for the cases when the traditional Z-Test does not work efficiently (e.g., when primitives arrive in a back-to-front order): we still have a second resort of using the final Z depth from the previous frame. Due to frame-to-frame coherence, if a fragment was occluded in the previous frame, it is highly likely that it will also be occluded in the current frame. However, this approach does not guarantee that fragments discarded in the Ω-Test will not be visible in the final image. It might happen that a fragment that does not pass the Ω-Test is finally visible (e.g., an object that suddenly appears in front of other objects) leading to a potential *error* in the tile. For such cases, our approach implements a simple error detection and correction mechanism right after the tile is shaded, with a small overhead, as it will be further detailed in Section IV-B.

A major characteristic of a TBR pipeline is that the working unit is a tile. After finishing a tile, all of its information is discarded, including the valuable Z-Buffer. If we want to keep a tile's Z-Buffer to be used in the next frame, it must be stored somewhere. Thus, we would need additional storage to keep the individual Z-Buffers of every tile in a frame, which could be a too large overhead. The structure used to save the information of all the Z values

of the entire previous frame is called Ω-Table. If we decided to store it on on-chip buffers the required amount of memory for a frame in Full-HD resolution (1920x1080 pixels) would be around 8 MBytes, which would contradict the TBR philosophy that encourages the use of small memories for better energy efficiency. A second solution would be to store the Ω-Table in DRAM. The drawback of the later would be the intensive use of memory because of the additional transfers before and after processing each tile, resulting in prohibitive energy costs (recall DRAM consumes more than 50 % of the baseline GPU's energy, as reported in Section I). We evaluated this second solution to quantitatively measure the impact of storing the Ω-Table in DRAM. Unfortunately, the net effect in the overall energy consumption was negative, since the additional DRAM accesses more than offset the benefits coming from the overshading reduction.

To efficiently cope with the storage needs associated to our approach, while not incurring in significant energy costs, the proposed solution consists of not storing all the Z values from the previous frame but a small set of representative ones. Even though this results in a loss of information, as we will describe next, we observed that just keeping a few representative values per tile was as efficient as keeping the complete tile's Z-Buffer. Obviously, there is a small number of induced errors for not using 100 % precise information but the mechanism for detecting and correcting errors (see Section IV-B for additional details) fix them whereas the incurred overhead from the correction phase pays off w.r.t. the energy we save from neither using large on-chip memories nor relying on DRAM for storing the previous frame's Z values. One alternative approach we have not evaluated is a hybrid implementation where Z-values are stored in DRAM and an on-chip memory is used as a cache.

There are multiple ways for compressing the Ω-Table or selecting a set of representative values. However, we avoided traditional compression algorithms because of their hardware complexity and energy cost. As cited before, the Ω-Test already generates some errors from the fact it relies on depths from the previous frame, which are later corrected, so we do not need to be totally precise with the information to use. A fast and simple algorithm that loses information can be more appropriate for our purposes than a complex algorithm that compresses without losses. Therefore, as a trade-off solution, we decided to make use of conventional *aggregate functions* (e.g., maximum, minimum, arithmetic mean) to select the set of representative Z values. Aggregate functions are simple to implement in hardware.

In addition to using aggregate functions, as we are compressing a 2-dimensional array, we also define a *coarsening factor* that represents the granularity level (or partitioning) we make to the Ω-Buffer. For example, assuming tiles of 4x4 pixels, a coarsening factor of 2x2 will break the tile down into 4 non-overlapping squares of 2x2 pixels. Then, each of these 2x2 squares will go through the aggregate function. The result will be a matrix of 2x2 elements containing the resulting values of the applied aggregate function. Summarizing, we go from a matrix of 4x4 Z values to one of 2x2 Z values, which reduces the required memory by a factor of 4x. To better illustrate this, Figure 4 depicts this compression scheme based on using a coarsening factor followed by an aggregate function, assuming tiles of 4x4 pixels.

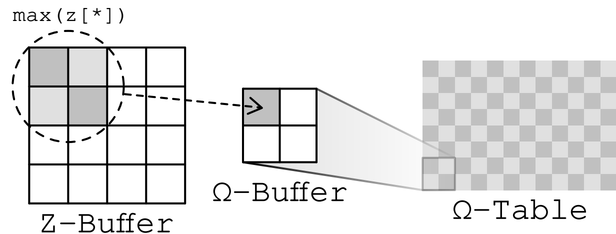

Fig. 4: Compression scheme overview for a 2x2 coarsening factor and considering tiles of 4x4 pixels.

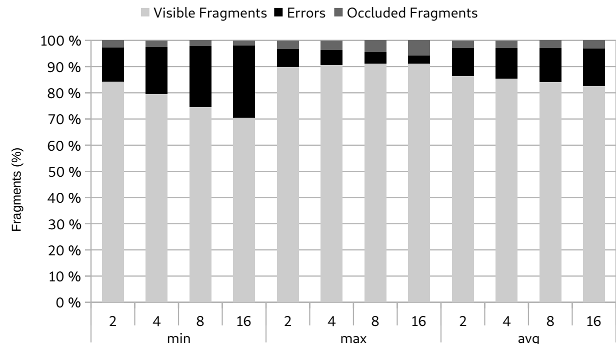

Fig. 5: Effect of using different coarsening factors (from 2x2 to 16x16) for 3 aggregate functions (min, max, average).

We have evaluated the effect of using different coarsening factors along with different aggregate functions (maximum, minimum and average). Figure 5 shows a preliminary study of the error rate with respect to the total amount of shaded fragments (visible and occluded). This preliminary study quantifies the errors that appear for the different compression schemes used for the Ω-Buffer.

Let us discuss first how the different aggregate functions behave. The *minimum* function keeps the fragments closer to the camera, so the Ω-Test is more restrictive and ends up discarding more fragments than needed. Overshading is reduced at the cost of generating too many errors, as it can be seen in Figure 5. On the other hand, using the *maximum* function makes the Ω-Test more permissive since we compare against the deepest Z of the tile. Overshading is not reduced as much but it produces as few errors as possible. Finally, using the *average* as the aggregate function leads to a trade-off between errors and overshading. As our goal is to generate as few errors as possible, due to the high overhead of the correction phase, we have chosen the *maximum* as the aggregate function for the Ω-Buffer, i.e., all the Z values of a group will be represented by the depth of the visible fragment most distant to the camera.

Regarding the coarsening factor, Figure 5 also shows that even going up to the highest one (16x16) does not incur a significant potential loss. As a re-

Tabla I: Ω-Table storage needs for different coarsening factors, assuming a 1280x720 screen resolution.

Coarsening level	Ω-Table size
1x1	3.51 MiB
2x2	900 KiB
4x4	225 KiB
8x8	56.25 KiB
16x16	14.06 KiB

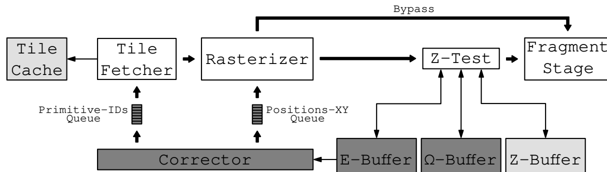

Fig. 6: Ω-Test implementation on a TBR architecture. Darkened boxes correspond to added stuff over the baseline design.

sult, we have chosen the highest coarsening factor (16x16) which in practice means that each tile will represented for only one Z value, in particular the *maximum* one. This supposes a reduction by a factor of 256x in the storage needs without hardly penalizing the potential. Table I shows the memory required by the Ω-Table depending on the coarsening factor for a common HD screen resolution (1280x720 pixels). As an example, assuming tiles of 16x16 pixels, there will be 80x45 tiles in the frame, or a total of 3600 tiles, each needing 4 Bytes for the aggregated Z value. This totals an amount of 14 KiB for storing the Ω-Table for the whole frame, which can be easily implemented as an on-chip buffer.

IV. Ω-Test Implementation Details

The proposed Ω-Test is mainly applied in the Early Depth Test stage of the graphics pipeline. This test is similar to the Z-Test, but instead of getting the depth to compare with from the Z-Buffer, it is provided by the Ω-Buffer. Figure 6 shows the modifications made to the base architecture (with a darker color to differentiate them) to implement our proposal.

The main advantage of our technique is that it starts with a *speculative* version of the Z-Buffer, called the Ω-Buffer, so it can remove many more fragments. The downside is that it might lead to some mistakes. They happen, for instance, when a primitive moves away from the camera. A particular fragment of such primitive will have a Z value in frame i which is kept in the Ω-Table to be used in frame $i+1$. In this particular case, since the fragment has moved away (resulting in a Z' greater greater than Z) it will pass the regular Z-Test (initialized to $-\infty$). However, it will fail the Ω-Test since the object has moved back (Z' ¿Z).

A second case that might lead to potential errors is the lateral movement of an object. In this case, as the object moves (e.g., from left to right) it hides a part of the background with its right-side border while making visible the background behind its left-side border. In this case, fragments from the background that were occluded in frame $i-1$ become visible in frame i, leading to potential errors in the final image.

With the aim of mitigating the amount of errors coming from the first case, i.e., objects that move away from the camera across consecutive frames, we use a technique which consists of applying a Delta (δ) margin to the Ω depths, as we will explain in next Section IV-A.

Finally, any remaining error, either coming from lateral movements or as a result of using an *aggregated* Z value for representing a whole tile, is solved by adding a final correction step detailed in Section IV-B.

A. Reducing Errors: Delta (δ) Margin

It is very common to have scenes in which most of the Z-Buffer remains intact. In this case, our technique acts in an ideal way because it does not produce errors, nor does it draw unnecessary fragments. However, there are other scenes in which objects move, not necessarily affecting all the tiles in a frame but a fraction of them. As cited before, a particular type of movement that might potentially lead to errors happen when objects move away from the camera.

To be able to tolerate such movements while reducing the amount of potential errors, we include a small safety margin for the Ω-Table, called delta (δ) margin. By doing this, we relax the Ω-Test condition since it is equivalent to slightly moving the depths of all the fragments a bit further. The rationale behind this δ margin is to be more permissive by not eliminating fragments that belong to objects that have slightly moved away from one frame to the next. By using this safety margin, the Ω-Test becomes more flexible and incurs in less errors. On the other hand, the amount of overshading is not significantly hurt because this δ margin is very small.

In essence, we are trading errors for overshading. I.e,. the δ margin helps reducing the amount of errors at the expense of not being able to avoid some overshading. To better understand the effect of using this safety margin, we have analyzed a wide range of δ values and measured how the amount of errors and the overshading factor are affected. It can be observed in Figure 7 that as we increase δ, it allows for more fragments to pass the Ω-Test, resulting in higher overshading. However, the number of errors is reduced. Contrarily, smaller δ values give less margin to depth changes in the Ω-Buffer, causing more errors but reducing overshading. Figure 7 shows that the best δ is 0.0005 for many games (recall Zs are normalized in the range [0,1] where 0 corresponds to the camera viewpoint and 1 represents the infinite). Note also that frame-to-frame coherence has a significant influence on δ, because the more coherence the smoother the movements will be and, therefore, smaller δ values will suffice.

However, using a static δ for all the games does not provide the best trade-off, given the high variability that can be found in games. To cope with this inter-frame variability, we have implemented a very simple dynamic scheme that defines a δ value for each fra-

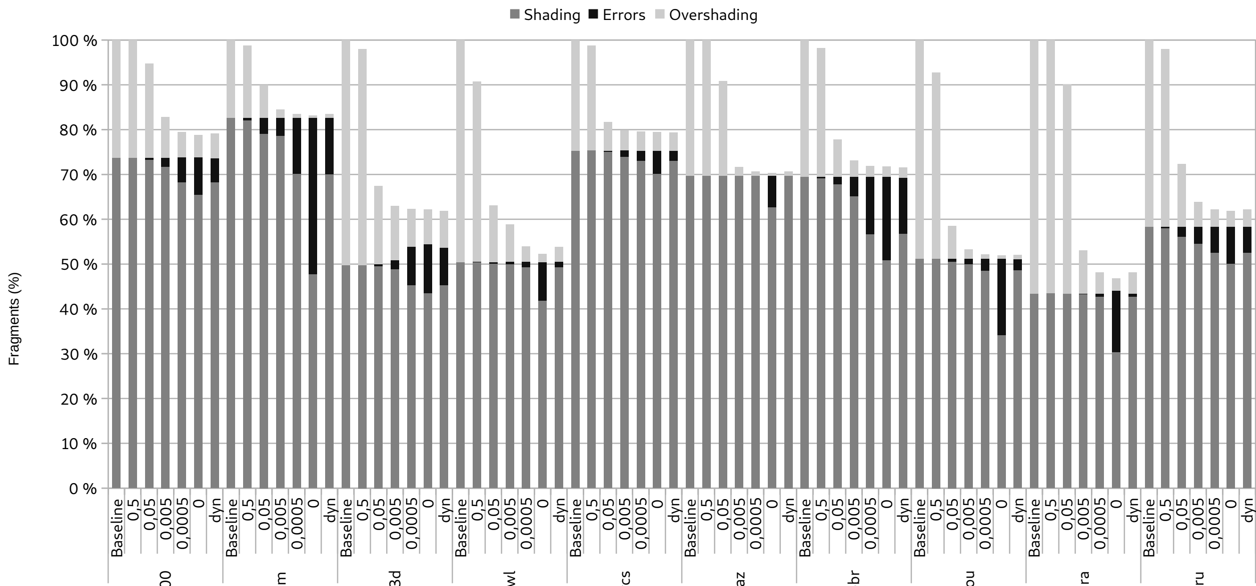

Fig. 7: Study of the ratio overshading/errors for several static δ margins (0.5, 0.05, 0.005, 0.0005 and 0). The last bar (dyn) corresponds to the dynamic δ implementation.

me, so that this safety margin is adapted depending on whether the objects within a frame move away or not. First, we made an experimental study to quantify the cost of correcting an error, and we measured that it is 3x higher than shading a fragment, so the adaptive scheme prioritizes reducing the amount of induced errors.

The adaptive technique changes δ based on frame-level overshading/error ratios. We define a cost function (Equation 1) whose inputs are the number of overshaded fragments and errors. In (1) wo is the weighted cost associated to overshading and we is the weighted cost associated to errors, always speaking in terms of energy. Finally, o and e represent the per-frame amount of overshading and errors.

$$cost(o,e) = wo * o + we * e \tag{1}$$

The scheme uses a table of eight δ values (0.0001, 0.0005, 0.001, 0.005, 0.01, 0.05, 0.1, 0.5); an index pointing to the current δ, a variable that indicates in which direction we move this index, and two global counters that account for the number of errors and overshading for the whole frame. The initial value for δ is set to 0.0005 (since it was the best static δ). The adaptive scheme operates as follows. When finishing rendering a frame the cost function (1) is evaluated and compared with that of the previous frame (held in a global register). If the current cost is higher, we change the direction of the index of the δ table, otherwise we move the index in the direction shown by the cited variable. If reaching the table limits, the index saturates.

To sum up, the dynamic δ scheme is able to effectively reduce the fraction of errors as intended. In particular, it can be observed in Figure 7 that the average fraction of errors (over the total amount fragments) is just 5.1 % thanks to using the dynamic δ scheme.

B. Error Detection and Correction

As the Ω-Test might lead to discarding a fragment which is actually visible in the final image, we need a mechanism to detect and correct mistakes. Mistakes are made in the Early Depth Test stage where it is checked whether a fragment must proceed or not to the Fragment Processors to be shaded. A fragment that passes the Z-Test but not the Ω-Test could be a potential error. However, note that this fragment can be *hidden* by another visible fragment rendered on top of it. In this case, the Ω-Test has avoided an undesired overshading case, saving some useless work. However, if no fragment is ever written in that position of the tile, a *gap* would be left in the final Color Buffer. Obviously, these induced gaps cannot be propagated to the Color Buffer and a corrective action must take place.

To keep track of the potential errors, some additional data structures are needed. In particular, we use a two-dimensional array called E-Buffer, with the same dimensions of a tile, where each cell represents a fragment of the tile and stores a primitive's *identifier.* Once the Z-Test is passed, we perform the Ω-Test. If the Ω-Test fails, we store the primitive ID in its corresponding position in the E-Buffer. This indicates the primitive *might* cause an error in that position. Otherwise, if the Ω-Test successes, we store in the E-Buffer an special ID (-1 in our case). Therefore, a final ID of -1 indicates that there is no error in that position. A global counter is also needed. This global counter is increased when a primitive ID is stored in the E-Buffer (overwriting a -1); and it is decreased when a -1 is stored (overwriting a valid primitive ID).

When all primitives have been rasterized and the Early Depth Test has no more fragments to process, the E-Buffer is in a final state and identifies where the final errors are located. At this point, the aforementioned global counter is checked, activating the

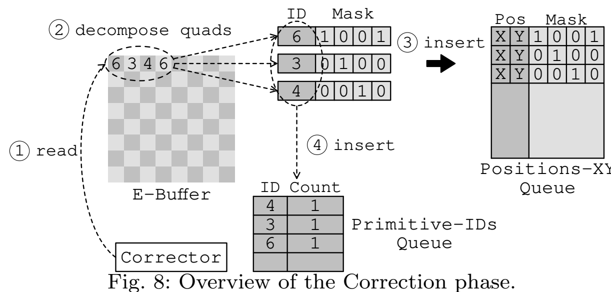

Fig. 8: Overview of the Correction phase.

correction mechanism if greater than 0 (i.e., the E-Buffer contains at least one error). It is important to note that we have to wait until all fragments have passed through the Early Depth Test stage.

To better illustrate the process, Figure 8 shows how the Corrector works. First, it reads quad fragments from the E-Buffer (step ①). For each different primitive ID of the quad, a visibility mask whose active bits are the positions it occupies within the quad is generated (step ②). This visibility mask determines which fragments of the quad are valid. Thus, a quad can generate four visibility masks if its four Primitive IDs are different (worst case). These masks along with the quad position are inserted into the Positions-XY queue (step ③). If all the valid primitives of the quad (according to the visibility mask) are equal (best case, and fortunately the most common) an internal counter is incremented, indicating the number of quads from the same primitive. When a different primitive is found, a new entry is inserted into the Primitive-IDs queue, containing both the ID of the previous primitive and the value of the internal counter (step ④).

As soon as there is a primitive in the Primitive-ID queue, the Tile Fetcher starts working to perform the corrections for the current tile. Under this correction mode, the Tile Fetcher rather than querying the Tile Cache for a new primitive, gets them from the Primitive-ID queue. Additionally, the number of quads of the same primitive is provided to indicate the Rasterizer how many entries (errors) from the Positions-XY queue will be corrected.

In the correction mode, the Rasterizer has a slightly different behaviour as well. In particular, only the fragments to be corrected are generated for a given primitive. I.e., if a triangle only has one erroneous pixel, this is the only fragment that will be generated. To do that, the Rasterizer calculates the barycentric coordinates of the first fragment and the X and Y increments. Note that the (X,Y) coordinates where the error is located are obtained from the Positions-XY queue, along with the visibility mask (refer to Figure 8). Then, the quad fragment is sent to the Fragment Processors for a proper shading. Also note these quads for correction do not go through the Early Depth Test stage because we certainly know they are visible.

After the correction phase, the graphics pipeline continues working as usual. When the tile is completely rendered and the Color Buffer is computed and flushed, the pipeline is ready to start with a new tile.

Finally, there is a challenging situation that happens when the Tile Fetcher finds a transparent primitive. A primitive is considered as transparent if its blending attribute is active, meaning that all the fragments from this primitive have to mix their rendered colors with the existing ones in the Color Buffer. Our Ω-Test proposal has to deal with these cases, otherwise, a blending error would be generated, mixing the transparent fragment with a black fragment. To overcome this situation, the correction phase is triggered as soon as the transparent fragment arrives into the Early Depth Test stage. When the correction of the errors for opaque primitives is done, the pipeline can continue processing the transparent fragment and the normal operation of the Tile Fetcher is resumed.

V. Evaluation Methodology

A. Simulator Infrastructure

We have used TEAPOT [14], a cycle-accurate simulator framework for GPUs based on Mali's Utgard architecture [15], to obtain performance and energy measurements. This simulator makes use of two well-known tools: McPAT for [16] for power estimation, and DRAMSim2 [17] for modelling DRAM and the memory controllers. The simulator is fed with trace files that were obtained by running the benchmarks either in a real smartphone or in an Android Virtual Device (AVD) [18]. The OpenGL [19] traces have been obtained with GAPID [20], a graphics debugger that allows to inspect the graphics commands of animated applications. In particular, the OpenGL trace is executed with the GAPID replay tool (`gapir`) over an instrumented Gallium Softpipe Driver [21] to obtain the final trace for TEAPOT. Table II shows the GPU simulation parameters, resembling the ARM Mali-450 GPU, that we have used to evaluate our proposal.

B. Benchmarks

Table III shows the benchmarks we have used in our evaluations. They correspond to games selected based on their popularity in number of downloads in the Google Play Store [22]. Note that we have only considered 3D games since our technique does not apply to 2D games.

VI. Experimental Results

Figure 9 shows the runtime speedup achieved by the Ω-Test. We can see that the proposed technique provides an average speedup of 10.6 %, with a maxium speedup of 17.2 % for Maze 3D. Although our technique dramatically reduces overshading, runtime is not affected in the same proportion. This is because of the overheads to correct errors generated by the Ω-Test and pipeline stalls to wait for that correction. Such overheads represent, on average, around 6 % of the total execution time.

In terms of energy consumption, as reported in Figure 10, the Ω-Test provides average savings of

Tabla II: GPU Simulation Parameters.

Baseline GPU Parameters	
Frequency	400 MHz
Voltage	1.0 V
Scale Integration	32 nm
Screen Resolution	1280x720
Tile Size	16x16 pixels
Main Memory	
Frequency	400 MHz
Voltage	1.5 V
Latency	50-100 cycles
Bandwidth	4 B/cycle (dual channel LPDDR3)
Size	1 GiB
Queues	
Vertex (Input & Output)	16 entries, 136 bytes/entry
Triangle & Tile	16 entries, 388 bytes/entry
Fragment	64 entries, 233 bytes/entry
Color	64 entries, 24 bytes/entry
Caches All of 64 bytes/line, 2-way associativity	
Vertex Cache	4 KiB, 1 bank, 1 cycle
Texture Caches (x4)	8 KiB, 1 bank, 1 cycle
Tile Cache	128 KiB, 8 banks, 1 cycle
L2 Cache	256 KiB, 8 banks, 2 cycles
Color Buffer	1 KiB, 1 bank, 1 cycle
Depth Buffer	1 KiB, 1 bank, 1 cycle
Non-programmable stages	
Primitive assembly	1 triangle/cycle
Rasterizer	1 attributes/cycle
Early Z test	8 in-flight quad-fragments
Programmable stages	
Vertex Processor	4 vertex processor
Fragment Processor	4 fragment processors
Ω-Test hardware	
Ω-Buffer	14 KiB
Positions-XY Queue	64 entries, 13 bytes/entry
Primitive-ID Queue	64 entries, 8 bytes/entry
E-Buffer	1 KiB
Corrector	4 quad-fragments/cycle

Tabla III: Evaluated benchmarks set.

Benchmark	Alias	Description	Downloads (M)
300	300	hack & slash	10-50
Captain America	cam	beat'em up	1-5
Sniper 3D Assassin	s3d	Shooter	100-500
Hot Wheels: Race Off	hwl	Racing	50-100
Vegas Crime Simulator	vcs	Sandbox & Crime	100-500
Maze 3D	maz	Labyrinth	10-50
Beach Buggy Racing	bbr	Racing	50-100
Counter Strike	cou	Shooter	10-50
Gravity	gra	Action	1-5
Temple Run	tru	Adventure arcade	100-500

15.7 %, and going up to 22.2 % in Beach Buggy Racing. Games such as Maze 3D, Vegas Crime Simulator, Gravity or the aforementioned Beach Buggy Racing get better results because of their textures sizes. Much of the cost of overshading comes from DRAM accesses derived from misses in the Texture Cache. A game with small textures is more likely to obtain a higher hit rate in the Texture Cache, therefore, reducing its overshading factor does not save as many accesses to main memory. Contrarily, games with complex, detailed and high-resolution textures, will exhibit a lower hit rate in Texture Cache, going more frequently to DRAM memory. Therefore, reducing the overshading factor in such games will have a much more noticeable impact.

Other benchmarks, such as 300 or Counter Strike,

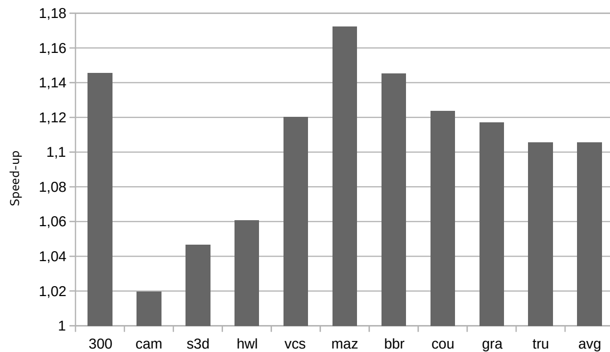

Fig. 9: Ω-Test speedup over the TBR baseline architecture.

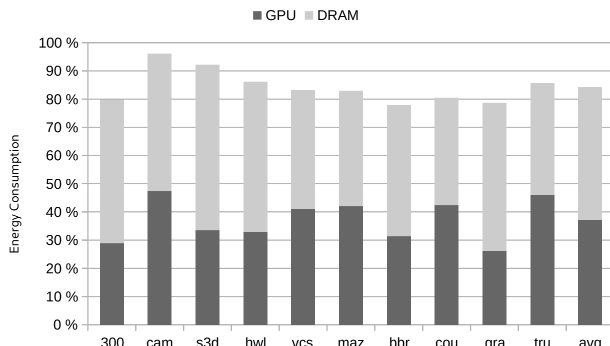

Fig. 10: Ω-Test energy consumption over the TBR baseline architecture.

obtain large energy savings (around 20 %), not because of the size or complexity of their textures, but because of the huge volume of overshading that is exposed to the baseline TBR architecture, and which our proposal is significantly removing.

As for the area overhead, the Ω-Test involves the use of additional hardware, as detailed in Section IV, whose area has been measured to be 2.46 mm^2, which represents 2.3 % of the total area of the GPU. Of all the structures introduced by the proposed Ω-Test, the largest is the Ω-Table (with 14.06 KiB) that translates to an area of 1.8 mm^2, which represents 1.71 % of the total area of the GPU. Finally, note that the energy consumption of the new structures have been properly modeled and accounted for in our experiments.

VII. Related Work

Z-Prepass [23] is a software technique capable of eliminate overshading caused by hidden surfaces that consists on two rendering passes. First, the entire geometry of the scene is calculated and rasterized with a *null* shader fragment, so that only the depth values are calculated and stored in the Z-Buffer. In the second pass, the depth values are in their final state, which allows the Z-Test to reduce the overshading of opaque surfaces to the minimum. However, the extra rendering pass introduces a high cost that is only offset by greater shading savings usually only possible on desktop applications with costly fragment shaders, which makes this technique not suitable for mobile GPUs.

Deferred Rendering (DR) [24] is able to discard the useless fragments of the final scene like Z-Prepass does. To do this, a Hidden Surface Removal (HSR)

phase is added to the pipeline just before the Early Depth Test stage. The HSR phase iterates over all the rasterized fragments just calculating their position and depth to build a complete Z-Buffer. Afterwards, the actual visible fragments are known and sent to the Fragments Processors. The main overhead of DR is the fact that all the primitives have to be rasterized twice, and so the fragments are processed twice as well: once for calculating the depths in the HSR phase, and again for calculating the rest of attributes in order to continue down the pipeline. This forces the designers to either increase the pressure over the existing hardware (with the subsequent degradation of the execution time and therefore of the energy consumption) or to include significant extra hardware to perform the extra computations of HSR (duplicate rasterizer, Z-Test, and Z-Buffer) [5]. Differently, our proposal introduces a small on-chip buffer and a fast Ω-Test.

Visibility Rendering Order (VRO) [7] is another HSR technique that optimizes the visibility problem by sorting the objects of a 3D scene. The order in which commands are drawn in a scene can have a significant impact on the GPU's final performance. A scene where the farthest objects are drawn first will incur a degradation in GPU performance since the Z-Buffer will not be able to remove most of the fragments it renders. VRO takes advantage of the frame-to-frame coherence to arrange objects in a front-to-back order, increasing the quad fragments discarded by the Z-Test. While VRO solves the visibility problem at the object level, our Ω-Test operates at the much finer fragment level, being able to overcome not only inter-object overshading but also intra-object overshading.

Hierarchical Z-Buffer Visibility [8] is another HSR technique that solves the problem of visibility by means of a much more complex Z-Buffer organization. Apart from frame-to-frame coherence, this technique takes advantage of object-space coherence and image-space coherence. To do that, it uses an octree to determine whole visible objects at once. They also use a pyramidal Z-Buffer. At the lowest level, they have the full-resolution Z-Buffer, while subsequent upper levels divide the resolution by 4, until reaching to the last level that has a single pixel. Authors claim to be more efficient when discarding fragments since they can eliminate groups of fragments at once instead of individually.

Early Visibility Resolution (EVR) [9] is a very recent HSR technique that speculatively resolves the visibility of objects in a scene before the Raster Pipeline. On the one hand it optimizes the Z-Test. As mentioned above, the order in which primitives arrive at the Raster Pipeline can have a noticeable impact on GPU performance. EVR compares the depth of each primitive against the farthest point resolved for each tile in the previous frame. Using that information, primitives are reordered in order to make the Z-Test capable of removing more fragments. On the other hand, it optimizes Rendering Elimination (RE) [25]. RE avoids rendering a tile if it has the same primitives as in the previous frame. The problem with RE is that a hidden primitive that changes over frames also results in a different signature, leading to a considerable loss of potential. EVR overcomes this situation by removing hidden primitives from the signature.

VIII. Conclusions

Overshading plays an important role in the performance and energy efficiency of mobile GPUs, and it is strongly related with the approach used to resolve the visibility of the different primitives. In this work we have proposed the Ω-Test, a novel microarchitecture technique that resolves visibility by using information of the Z-Buffer from the previous frame. We have shown that our approach is much more effective for removing overshading than the traditional Z-Test, which only uses information from the current frame and whose Z-Buffer must be built from scratch for every new frame.

Our approach relies on frame to frame coherence, however, an unexpected depth change in a primitive could potentially lead to an error in the final rendered image. We have included an error detection and correction mechanism to fix the small amount of errors that can appear in certain tiles. Finally, to dramatically reduce the storage needs of the underlying Ω-Buffer, we have also implemented a coarsening mechanism along with the use of an aggregate function, which is highly effective and hardly impacts accuracy. Overall, the Ω-Test reduces the average overshading of scenes by 33.8 %, which results in an average 15.7 % energy reduction in addition to an average speedup of 10.6 % for a set of representative applications.

Acknowledgements

This work has been supported by the the CoCoUnit ERC Advanced Grant of the EU's Horizon 2020 program (grant No 833057), the Spanish State Research Agency under grant TIN2016-75344-R (AEI/FEDER, EU) and the ICREA Academia program. D. Corbalán-Navarro has been supported by a PhD research fellowship from the University of Murcia.

Referencias

[1] M Shebanow, "An evolution of mobile graphics," *Keynote talk at High Performance Graphics*, 2013.

[2] Shruti Patil, Yeseong Kim, Kunal Korgaonkar, Ibrahim Awwal, and Tajana S Rosing, "Characterization of user's behavior variations for design of replayable mobile workloads," in *International Conference on Mobile Computing, Applications, and Services*. Springer, 2015, pp. 51–70.

[3] Jieun Lim, Nagesh B Lakshminarayana, Hyesoon Kim, William Song, Sudhakar Yalamanchili, and Wonyong Sung, "Power modeling for gpu architectures using mcpat," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 19, no. 3, pp. 26, 2014.

[4] Jeff Pool, *Energy-precision tradeoffs in the graphics pipeline*, Ph.D. thesis, The University of North Carolina at Chapel Hill, 2012.

[5] Enrique De Lucas, *Reducing redundancy of real time computer graphics in mobile systems*, Ph.D. thesis, Universitat Politècnica de Catalunya, 2018.

[6] Jiří Bittner and Peter Wonka, "Visibility in computer graphics," *Environment and Planning B: Planning and Design*, vol. 30, no. 5, pp. 729–755, 2003.

[7] Enrique De Lucas, Pedro Marcuello, Joan-Manuel Parcerisa, and Antonio González, "Visibility rendering order: Improving energy efficiency on mobile gpus through frame coherence," *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 2, pp. 473–485, 2018.

[8] Ned Greene, Michael Kass, and Gavin Miller, "Hierarchical z-buffer visibility," in *Proceedings of the 20th annual conference on Computer graphics and interactive techniques*. ACM, 1993, pp. 231–238.

[9] Martí Anglada, Enrique de Lucas, Joan-Manuel Parcerisa, Juan L Aragón, and Antonio González, "Early visibility resolution for removing ineffectual computations in the graphics pipeline," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2019, pp. 635–646.

[10] Tomas Akenine-Moller, Eric Haines, and Naty Hoffman, *Real-time rendering*, AK Peters/CRC Press, 2019.

[11] Tomas Akenine-Moller and Jacob Strom, "Graphics processing units for handhelds," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 779–789, 2008.

[12] Harold Hubschman et al., "Frame-to-frame coherence and the hidden surface computation: constraints for a convex world," *ACM Trans. on Graphics*, vol. 1, no. 2, pp. 129–162, 1982.

[13] Andrew Wilson, Ketan Mayer-Patel, and Dinesh Manocha, "Spatially-encoded far-field representations for interactive walkthroughs," in *Proceedings of the ninth ACM international conference on Multimedia*. ACM, 2001, pp. 348–357.

[14] Jose-Maria Arnau, Joan-Manuel Parcerisa, and Polychronis Xekalakis, "Teapot: a toolset for evaluating performance, power and image quality on mobile graphics systems," in *Proceedings of the 27th International ACM Conference on Supercomputing*. ACM, 2013, pp. 37–46.

[15] "Arm mali-450 gpu," `https://developer.arm.com/products/graphics-and-multimedia/mali-gpus/mali-450-gpu`, accessed August 2019.

[16] Sheng Li, Jung Ho Ahn, Richard D Strong, Jay B Brockman, Dean M Tullsen, and Norman P Jouppi, "Mcpat: an integrated power, area, and timing modeling framework for multicore and manycore architectures," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 2009, pp. 469–480.

[17] Paul Rosenfeld, Elliott Cooper-Balis, and Bruce Jacob, "Dramsim2: A cycle accurate memory system simulator," *IEEE computer architecture letters*, vol. 10, no. 1, pp. 16–19, 2011.

[18] "Android sdk," `https://developer.android.com/studio`, accessed August 2019.

[19] Mark Segal and Kurt Akeley, "The opengl graphics system: A specification (version 1.1)," 1999.

[20] "Gapid," `https://developers.google.com/vr/develop/unity/gapid`, accessed August 2019.

[21] "Gallium3d," `https://www.freedesktop.org/wiki/Software/gallium`, accessed August 2019.

[22] "Google play," `https://play.google.com`, accessed August 2019.

[23] Christopher A Burns and Warren A Hunt, "The visibility buffer: a cache-friendly approach to deferred shading," *Journal of Computer Graphics Techniques (JCGT)*, vol. 2, no. 2, pp. 55–69, 2013.

[24] Imagination Technologies Limited, "PowerVR Hardware. architecture overview for developers," `http://cdn.imgtec.com/sdk-documentation/PowerVR+Hardware.Architecture+Overview+for+Developers.pdf`, accessed August 2019.

[25] Martí Anglada, Enrique de Lucas, Joan-Manuel Parcerisa, Juan L Aragón, Pedro Marcuello, and Antonio González, "Rendering elimination: Early discard of redundant tiles in the graphics pipeline," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2019, pp. 623–634.

[26] Zhenghong Wang, "Dynamically optimized deferred rendering pipeline," Dec. 12 2017, US Patent 9,842,428.

[27] Andrew S Glassner, *An introduction to ray tracing*, Elsevier, 1989.

[28] Jeffrey Hao CHU, Subrato Kumar De, Dexter Tamio Chun, Bohuslav Rychlik, and Richard Alan STEWART, "Transaction elimination using metadata," Oct. 30 2018, US Patent App. 10/114,585.

[29] Harry Nyquist, "Certain topics in telegraph transmission theory," *Transactions of the American Institute of Electrical Engineers*, vol. 47, no. 2, pp. 617–644, 1928.

[30] Nasir Ahmed, T_ Natarajan, and Kamisetty R Rao, "Discrete cosine transform," *IEEE transactions on Computers*, vol. 100, no. 1, pp. 90–93, 1974.

[31] Zhou Wang, Alan C Bovik, Hamid R Sheikh, Eero P Simoncelli, et al., "Image quality assessment: from error visibility to structural similarity," *IEEE transactions on image processing*, vol. 13, no. 4, pp. 600–612, 2004.

[32] "Designware 2d dct," `https://www.synopsys.com/dw/ipdir.php?c=DW_dct_2d`, accessed August 2019.

[33] "Synopsys," `https://www.synopsys.com`, accessed August 2019.

[34] Kurt Akeley, "Reality engine graphics," in *Proceedings of the 20th annual conference on Computer graphics and interactive techniques*. ACM, 1993, pp. 109–116.

[35] Lei Yang, Diego Nehab, Pedro V Sander, Pitchaya Sitthi-amorn, Jason Lawrence, and Hugues Hoppe, "Amortized supersampling," in *ACM Transactions on Graphics (TOG)*. ACM, 2009, vol. 28, p. 135.

[36] Ian Mallett and Cem Yuksel, "Deferred adaptive compute shading," in *Proceedings of the Conference on High-Performance Graphics*. ACM, 2018, p. 3.

[37] Rahul Sathe and Tomas Akenine-Möller, "Pixel merge unit.," in *Eurographics (Short Papers)*, 2015, pp. 53–56.

[38] Karthik Vaidyanathan, Marco Salvi, Robert Toth, Tim Foley, Tomas Akenine-Möller, Jim Nilsson, Jacob Munkberg, Jon Hasselgren, Masamichi Sugihara, Petrik Clarberg, et al., "Coarse pixel shading," in *Proceedings of High Performance Graphics*. Eurographics Association, 2014, pp. 9–18.

[39] Yong He, Yan Gu, and Kayvon Fatahalian, "Extending the graphics pipeline with adaptive, multi-rate shading," *ACM Transactions on Graphics (TOG)*, vol. 33, no. 4, pp. 142, 2014.

[40] Karthik Vaidyanathan, Robert Toth, Marco Salvi, Solomon Boulos, and Aaron Lefohn, "Adaptive image space shading for motion and defocus blur," in *Proceedings of the Fourth ACM SIGGRAPH/Eurographics conference on High-Performance Graphics*. Eurographics Association, 2012, pp. 13–21.

[41] Magnus Andersson, Jon Hasselgren, Robert Toth, and Tomas Akenine-Möiler, "Adaptive texture space shading for stochastic rendering," in *Computer Graphics Forum*. Wiley Online Library, 2014, vol. 33, pp. 341–350.

[42] Karl E Hillesland and JC Yang, "Texel shading," in *Proceedings of the 37th Annual Conference of the European Association for Computer Graphics: Short Papers*. Eurographics Association, 2016, pp. 73–76.

[43] Christopher A Burns, Kayvon Fatahalian, and William R Mark, "A lazy object-space shading architecture with decoupled sampling," in *Proceedings of the Conference on High Performance Graphics*. Eurographics Association, 2010, pp. 19–28.

[44] Petrik Clarberg, Robert Toth, Jon Hasselgren, Jim Nilsson, and Tomas Akenine-Möller, "Amfs: adaptive multi-frequency shading for future graphics processors," *ACM Transactions on Graphics (TOG)*, vol. 33, no. 4, pp. 141, 2014.

[45] "Nvidia gpu turing architecture," `https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf`, accessed August 2019.

[46] Enrique De Lucas, "Reducing redundancy of real time computer graphics in mobile systems," *PhD Thesis*, 2018.

[47] Francesco Marcelloni and Massimo Vecchio, "A simple algorithm for data compression in wireless sensor networks," *IEEE communications letters*, vol. 12, no. 6, pp. 411–413, 2008.