Triangle Dropping: An Occluded-geometry Predictor for Energy-efficient Mobile GPUs

DAVID CORBALÁN-NAVARRO and JUAN L. ARAGÓN, Deptartamento de Ingeniería y Tecnología de Computadores, Universidad de Murcia, Spain
MARTÍ ANGLADA, JOAN-MANUEL PARCERISA, and ANTONIO GONZÁLEZ, Departament d'Arquitectura de Computadors, Universitat Politécnica de Catalunya, Spain

This article proposes a novel micro-architecture approach for mobile GPUs aimed at early removing the occluded geometry in a scene by leveraging frame-to-frame coherence, thus reducing the overall energy consumption. Mobile GPUs commonly implement a Tile-Based Rendering (TBR) architecture that differentiates two main phases: the *Geometry Pipeline*, where all the geometry of a scene is processed; and the *Raster Pipeline*, where primitives are rendered in a framebuffer. After the Geometry Pipeline, only non-culled primitives inside the camera's frustum are stored into the *Parameter Buffer*, a data structure stored in DRAM. However, among the non-culled primitives there is a significant amount that are rendered but non-visible *at all*, resulting in useless computations. On average, 60% of those primitives are completely occluded in our benchmarks. Despite TBR architectures use on-chip caches for the Parameter Buffer, about 46% of the DRAM traffic still comes from accesses to such buffer. The proposed *Triangle Dropping* technique leverages the visibility information computed along the Raster Pipeline to predict the primitives' visibility in the next frame to *early* discard those that will be totally occluded, drastically reducing Parameter Buffer accesses. On average, our approach achieves overall 14.5% energy savings, 28.2% energy-delay product savings, and a speedup of 20.2%.

CCS Concepts: • **Hardware → Power estimation and optimization**; • **Computer systems organization → Multicore architectures**; *Special purpose systems;*

The final publication is available at ACM via http://dx.doi.org/10.1145/3527861

1 INTRODUCTION

In the past years, the use of mobile devices such as smartphones, tablets, and smartwatches has increased exponentially and they have become an essential part of our daily life. This has been fueled by a plethora of applications that these devices can be employed for, video games being one of the most common usages. While a few years ago dedicated video-game consoles were the mainstream devices for playing games, it is nowadays a common activity performed on mobile devices, thanks to their improved performance, energy efficiency, and advanced graphics capabilities. Mobile **Systems on a Chip (SoCs)** are heterogeneous architectures that integrate a dedicated **GPU (Graphics Processing Unit)** whose energy efficiency has a huge impact on the whole system autonomy. However, users demand games with more complex graphics, and the more complex they are, the more battery is drained and the more heat is generated. Therefore, the SoC energy efficiency is a critical factor in mobile devices to improve the user experience [46, 53].

The vast majority of graphics workloads, especially 3D ones, are rendered by the GPU, whereas the CPU is mostly in charge of non-graphics tasks such as the main game loop, the user IO control interface, or calculating object/character trajectories. Apart from executing graphics workloads, GPUs have also been used over the past years for executing general-purpose programs and application-specific codes, a.k.a. GPGPUs, due to their high capabilities to perform massive-parallel computations [20, 32, 40, 42, 43, 55, 60]. To process the high complexity of graphics scenes, there are some graphics APIs that allow for a high-level CPU-GPU communication, being OpenGL [54], Vulkan [52], and DirectX [11] the most widely used APIs. In the particular case of OpenGL, a user-defined program executed in the CPU sends commands to the GPU. These commands carry information about the objects to be rendered. These object models are composed of primitives that in turn are made up of vertices. A vertex contains information not only about its position but also about its color, normal vector, texture coordinates to be mapped on, and other user-defined parameters. This huge amount of information, which in modern 3D games comes from hundreds of thousands of vertices and primitives, is commonly referred to as the *geometry* of the game.

At a high level, to render a scene, vertices are first transformed to screen coordinates by executing user-defined programs in the GPU (called *vertex shaders*). Then, all the primitives of each object are split into triangles that are later discretized (rasterized) into fragments. These fragments are later processed by executing user-defined programs in the GPU (called *fragment shaders*) to determine their final color values. A fragment shader accesses the textures to be mapped and applies a shading and a lighting model to every fragment to render a more realistic image. The resulting color value of a fragment is finally blended with other color values previously calculated and stored in the same position of the color buffer (to properly handle transparencies) leading to the final output image.

Previous studies have shown that a main contributor to the energy consumed on a mobile SoC is the GPU [37, 48] due to the huge number of computations and, especially, the DRAM accesses that are performed to render a frame. Figure 1 shows the power breakdown for a typical mobile GPU (built in a 22 nm technology process). It can be observed that 73% of the power is dissipated by accesses to the main memory. Therefore, reducing the number of DRAM accesses is critical to achieve significant energy savings.

Mobile GPUs commonly implement a **Tile-Based Rendering (TBR)** pipeline organization (further detailed in Section 2) to reduce DRAM accesses. To achieve that, a TBR architecture partitions the screen into small tiles and renders each tile in sequence onto a small on-chip color buffer, instead of accessing the full-screen framebuffer in main memory. The on-chip tile-sized color buffer is written into main memory at once, right after the tile is completely rendered. The downside

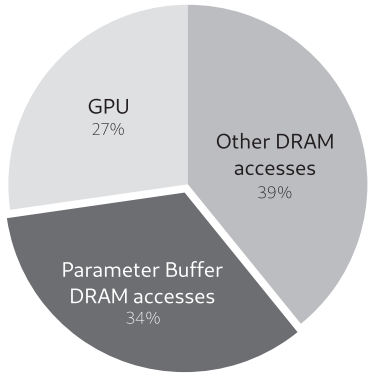

Fig. 1. Power breakdown of a typical mobile Tile-Based Rendering GPU.

of TBR is that it requires processing the scene geometry first, in the so-called Geometry Pipeline (refer to Figure 4). Then, it sorts all the primitives of the frame into tiles and stores them in a data structure known as the *Parameter Buffer*, which is kept in the off-chip DRAM because of its large size. Next, all the processed geometry data is fetched again, tile by tile, to be processed by both the Raster Pipeline and the **Hidden Surface Removal (HSR)** phase. Even though an on-chip Tile Cache is used to minimize the amount of DRAM accesses to the Parameter Buffer, due to the access pattern of the Tiling Engine, which processes tiles one-by-one, it is difficult to achieve a good reuse rate, and many accesses end up going to DRAM (in our benchmarks, the average miss rate of the Tile Cache is 80.81% for writes and 26.17% for reads).

However, there are a large fraction of primitives that after being rendered are fully occluded in a scene and consume a significant amount of computing and hardware resources to finally discard all the work done. To illustrate the impact of the geometry that is processed but eventually occluded in a scene, Figure 2 shows a wireframe representation of a portion of a scene (from one of the evaluated games—Hot Wheels) split into two parts. The left side shows *all* the triangles that fall inside the camera frustum view[1] and are front-facing, regardless of their visibility (a total of 27,554 triangles in the whole frame). The right side depicts the same scene showing only the triangles that are visible, either totally or partially (a total of 10,439 in the whole frame). In this example, 61.2% of the triangles that are processed (i.e., rasterized and rendered) are finally occluded. It is also important to note that, in terms of geometry, the cost of processing a triangle is the same regardless of its size on the screen or how far/close it is from the camera view.

To provide a more quantitative insight of the amount of visible/hidden geometry, Figure 3 shows a breakdown of the primitives (triangles) in a frame differentiating those that fall *outside* the camera frustum view (which are directly discarded by our baseline GPU) from those that fall *inside*. For the latter, the figure also differentiates primitives that are back-face culled (red stack—also discarded on our baseline GPU) and those primitives that are rasterized and passed to the Fragment Processors for shading (yellow and orange stacks), which account for 37.7% of the primitives on average. From those, it can be observed that 60.1% are fully occluded on average (orange stack).

This article proposes *Triangle Dropping*, a novel micro-architecture approach implemented in the graphics pipeline of a mobile GPU that is able to discard the fully occluded primitives of a frame early enough that they are not even written nor fetched to/from the Parameter Buffer, thus significantly reducing many DRAM accesses that would otherwise waste time and energy.

[1]Defined as the volume inside the six planes that delimit the visible space of a viewpoint.

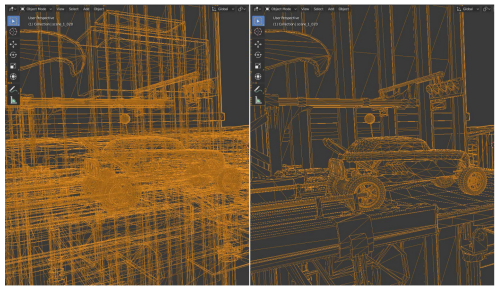

Fig. 2. Wireframe representation of a scene showing the amount of occluded geometry within the frustum. The left side shows *all* the front-facing triangles that fall inside the frustum (27,554), regardless of their final visibility. The right side only shows the triangles that are totally or partially visible (10,439).

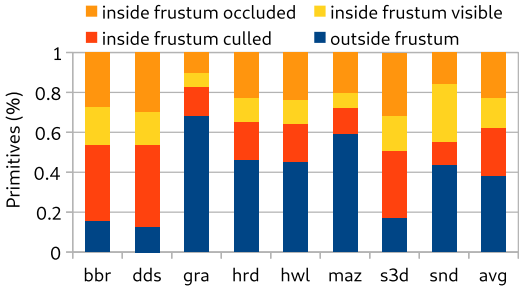

Fig. 3. Breakdown of primitives for the evaluated benchmarks. Triangle Dropping attacks the upper stack (orange), corresponding to primitives that fall inside the frustum but are eventually occluded.

Furthermore, as a positive side-effect, any further computations and memory accesses associated with the discarded primitives are also eliminated (rasterization, early-depth test, etc.). By leveraging frame-to-frame coherence [33, 58], our approach uses visibility information from the *previous* frame to predict the geometry that will be occluded in the *current* frame. To the best of our knowledge, Triangle Dropping is the first approach aimed at eliminating useless GPU activity derived from processing fully occluded primitives at such an early stage in the graphics pipeline.

The main contributions of this work are the following:

- We propose a hardware approach that early removes occluded geometry in a scene, reducing both the execution time and the energy consumption.
- We show a practical implementation of such a mechanism on top of a **Tile-Based Deferred Rendering (TBDR)** architecture, demonstrating that it can eliminate a significant amount of DRAM writes/reads into the Parameter Buffer.
- We evaluate Triangle Dropping with a set of commercial and popular games. Experimental results show average global energy savings of 14.5% in addition to an average speedup of 20.2%.

The rest of the article is organized as follows: Section 2 provides some background on Tile-Based Rendering GPU architectures. Section 3 explains the proposed Triangle Dropping approach, while Section 4 provides further implementation details. Section 5 describes our evaluation methodology, including an evaluation of the image and video quality, and Section 6 discusses the experimental results in terms of energy savings and performance improvement. Section 2.3 reviews the related work, and finally, Section 7 summarizes the main conclusions of the work.

2 BACKGROUND AND RELATED WORK

2.1 Tile-based Rendering Graphics Pipeline

As mentioned before, **Tile-Based Rendering (TBR)** architectures are widely used in mobile GPUs because of their lower power consumption [2]. The key aspect of TBR is the working set at which it operates. The screen area is divided into small regions of a fixed size called *tiles*, which allows to perform many operations on smaller on-chip structures avoiding costly accesses to off-chip DRAM memory. A tile is just a square chunk of the screen, generally 32×32 pixels, which is small enough to keep all its related data on-chip [5]. Unlike TBR, an **Immediate-Mode**

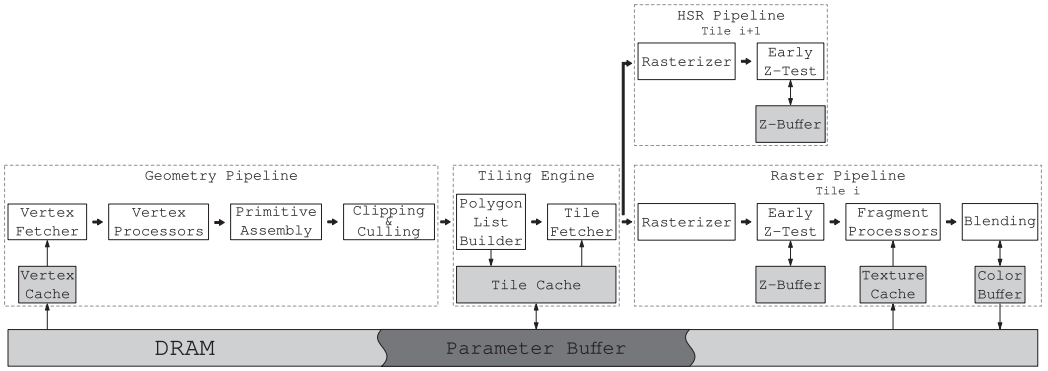

Fig. 4. Overview of the graphics pipeline of a TBDR GPU.

Rendering (IMR) graphics pipeline operates over the whole frame and it is the common architecture design for high-end GPUs, in which high performance is the main goal rather than low energy consumption.

The graphics pipeline of a TBR architecture is split into two main phases, the Geometry Pipeline and the Raster Pipeline, as depicted in Figure 4, which shows the main stages of the graphics pipeline. It is worth noting that both phases are serialized, with the Tiling Engine acting as a mediator in between. This serialization is mandatory, since tile-based processing requires all the geometry to be processed first to be able to determine which primitives belong to each tile. Only afterward, the rasterization and rendering of fragments are done on a per-tile basis.

The Geometry Pipeline performs all the geometry-related tasks of the scene to be rendered. The rendering starts with GPU commands, either *state commands* or *draw calls*. The first ones prepare the GPU by changing the internal OpenGL state machine. They are used for tasks such as CPU-GPU memory transfers (e.g., loading textures and object models), compiling a shader program, or setting some rendering parameters. However, if the command is a *draw call*, then the Vertex Fetcher is triggered. Vertices are fetched from the Vertex Cache and sent to the Vertex Processors, which execute user-defined programs called *vertex shaders* to transform the vertices and map them to the camera view plane. Once processed, vertices are passed over to the Primitive Assembly, which generates the corresponding primitives (e.g., triangles). These triangles pass through the Back-face Culling stage where they may be discarded accordingly. Next, the Clipping stage determines the triangles that are completely outside the *frustum view* to be also discarded. However, when a triangle partially overlaps the frustum, it is split into smaller triangles that entirely fall inside. The triangles resulting after the Clipping stage are assigned to the tiles they overlap by the Polygon List Builder. Finally, all the primitives that overlap each tile are stored in the Parameter Buffer (refer to Section 2.2 for further details).

Once all the *draw calls* are processed in the Geometry Pipeline, the rasterization and rendering of the scene are performed in the Raster Pipeline, tile-by-tile. The Tile Fetcher fetches primitives for a given tile and sends their attributes to a Raster Unit where triangles are discretized into pixel-sized units called *fragments*. Fragments pass to the Early Z-Test stage, where their depth values are tested against those from visible fragments processed so far, and those depths are stored in the Z-Buffer. A fragment that passes the Z-Test proceeds to the Fragment Processors to be shaded and textured. Otherwise, the fragment is invisible, and thus, discarded. Fragment Processors execute user-defined programs called *fragment shaders* to transform fragments' attributes into final colors. These colors are later processed by the Blending Unit, which blends them with those already in the Color Buffer, according to a blending function. When all the primitives of a tile are processed, the

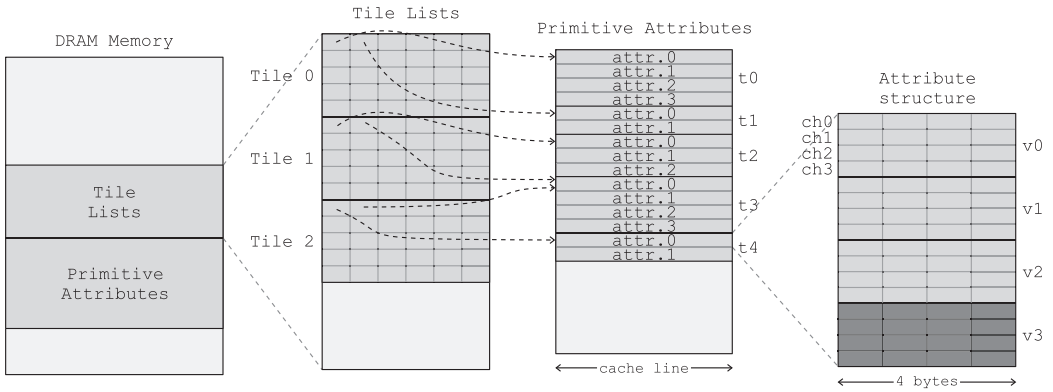

Fig. 5. Internal structure of a typical Parameter Buffer.

per-tile Color Buffer is flushed into the main memory in the corresponding render target. Finally, when all the tiles of the main render target are processed, the frame is ready to be visualized.

2.2 Parameter Buffer and Tiling Engine

The Parameter Buffer is a software structure held in main memory (as depicted in Figure 4) where all the scene's transformed geometry is stored on a per-tile basis. Figure 5 shows the internal structure of a typical implementation of the Parameter Buffer. This structure is split into two areas: the *Tile Lists* and the *Primitive Attributes*. The Tile Lists area stores, for each tile, a list of pointers to the primitives that overlap that tile. A primitive consists of a set of attributes that defines it (e.g., vertex coordinates, normal vectors, texture coordinates, and color information), which are located in the Primitive Attributes area. Note that a single primitive may belong (overlap) to more than one tile, which requires replicating the attribute pointers on each of those tiles but not the attributes themselves that are stored only once in the Primitive Attributes area.

Attributes are stored on a triangle basis, so each attribute contains information for three vertices ($v0, v1, v2$) plus a phantom vertex ($v3$) to make the attribute size equal to a cache line (64 Bytes). Each vertex contains four channels (storing either RGBA or XYZW information) of four Bytes each one (rightmost part of Figure 5).

The Tiling Engine, during the Polygon List Builder stage, fills up the Parameter Buffer once the Geometry Pipeline has ended. When a primitive arrives at the Polygon List Builder, its attributes are written into the Primitive Attributes area. The pointer to this primitive is also written into each tile that the primitive overlaps, i.e., in the Tile Lists area. When all the geometry has been sorted into tiles, the Raster Pipeline begins. For that, the Tile Fetcher schedules tiles to be rasterized and rendered one-by-one. For each tile, the Tile Cache is queried and upon a miss, the Parameter Buffer is accessed so all the information about the primitives that overlap the tile is read and cached.

2.3 Related Work

A different approach to reduce the number of primitives is followed by *mesh simplification* algorithms (a.k.a. *mesh decimation*). Their key idea is to render simplified versions of the 3D models with different ***level of detail*** **(LOD)** according to problem-specific restrictions. However, due to its complexity, *mesh simplification* is applied at the application level (or it is already pre-calculated) and not as a transparent on-the-fly hardware approach.

A very early proposal in this field was presented in Reference [51], whose main goal was to reduce the overall number of triangles in a dense mesh without hurting its topology. A

vertex-clustering and view-dependent method is proposed in Reference [50]. Another decimation approach, which prioritizes the mesh curvatures, is presented in Reference [29]. In Reference [3], an edge-collapse approach is presented, which tags candidate vertices to be deleted and then clusters them in pairs to collapse their common edges into a single vertex. An out-of-core GPU-based approach, based on vertex clustering and aimed at processing large polyhedral datasets, is presented in Reference [39]. An improvement presented in Reference [13] uses octree structures and relies on probabilistic methods to improve models quality. Another out-of-core GPU-based approach is presented in Reference [47] that also leverages frame-to-frame coherence. It performs edge collapse operations to simplify meshes.

A hybrid CPU-GPU implementation is presented in Reference [30]. In Reference [41], geometry shaders that implement pre-computed simplification algorithms are dynamically applied to meshes to do a tailored simplification. However, this method is *view-dependent*, which means that the simplification is performed with regards to the viewer perspective, so topology can be better preserved while reducing the same number of primitives. An OpenCL implementation of an edge-collapse simplification method is presented in Reference [45]. In Reference [31], an *energy function* that represents the compaction of a dense mesh is minimized, thus, giving a mesh with a similar topology but with fewer vertices than the original one. Again, this approach is intrinsically a non-real-time mechanism due to its complexity. Finally, the work in Reference [9] makes a comparison between different mesh simplification algorithms. Another survey on mesh simplification methods can be found in Reference [44]. Note, however, that all these approaches are software-based solutions aimed at reducing the overall number of triangles to be processed by the graphics pipeline and cannot be directly compared with Triangle Dropping.

The use of *occlusion queries* [1] also allows reducing the number of primitives to be processed by the graphics pipeline. These queries are explicitly inserted by the programmer before the OpenGL draw call of any command and tell the programmer whether the entire object is visible or not. This is achieved by pre-rasterizing the command's primitives and checking that all fragments fail their Z-Test, therefore, incurring a high overhead at the application level. A software-based occlusion culling algorithm is presented in Reference [34]. The process consists of locating *occluder* objects, then, a *shadow frustum* is computed for each one so objects that completely fit inside these volumes are removed. Another occlusion culling technique is proposed in Reference [15].

It is important to note that Triangle Dropping operates on top of these software-based approaches, being totally orthogonal to them. Triangle Dropping is fully hardware-based, and hence, programmer-agnostic. The source code for the vertex and/or fragments shaders is not needed and it can be implemented without any programmer intervention.

To the best of our knowledge, there are no purely hardware-based mechanisms like Triangle Dropping aimed at eliminating the occluded geometry of a scene in a programmer-agnostic manner, neither for mobile GPUs nor for high-end desktop GPUs. The most similar approach is the use of occlusion queries that, while it is true they can be implemented with some hardware support [59], it is still the programmer's responsibility to use them properly at the application level.

Nevertheless, as all the evaluated benchmarks are commercial games (whose source code is not publicly available), we assume that they already exploit all the GPU capabilities, including a potential implementation of occlusion queries, at the application level to get the best performance. In this sense, all the benefits of Triangle Dropping reported in this work are on top of any application-level optimization. Furthermore, an OpenGL recommendation is to issue as few commands as possible. An example of this can be seen in the **Hot Wheels benchmark (*hwl*)**, where all the buildings in the background are grouped into a single command, i.e., a single OpenGL draw call. As such, it is highly likely that a portion (even if small) of that big single command is visible in the current frame, making occlusion queries not able to achieve any benefit, as opposed to Triangle Dropping.

This is due to the much smaller granularity at which Triangle Dropping operates (primitive level), whereas occlusion queries are coarsely performed at the whole command (object) level. Patents cited in References [14, 26, 27, 35] also address the occlusion culling problem. Nevertheless, none of these approaches leverages frame-to-frame coherence to early discard occluded geometry. Patents cited in References [8, 16, 17, 25] also use Hidden Surface Removal to improve the performance of the visibility determination.

Finally, other techniques aimed at reducing overdraw have been proposed. **Visibility Rendering Order (VRO)** was presented in Reference [12] and reduces overdraw by re-ordering all the commands in the scene on-the-fly, so they are rendered in front-to-back order, allowing the Early-Z Test to eliminate more fragments than the baseline GPU. Differently, **Early Visibility Resolution (EVR)** [4] addresses the overdraw problem by avoiding the rendering of tiles that do not change from one frame to the next. To do that, it generates a per-tile signature based on the attributes of the primitives that overlap that tile. However, Omega-Test [10] works at a finer granularity, i.e., at the fragment level. To do so, Omega-Test creates a summarized version of the Z-Buffer from the previous frame to be used in the current frame as a better starting point, thus allowing the Z-Test unit to more efficiently discard fragments. Nevertheless, recall that overdraw reduction is not a key goal of Triangle Dropping, as mentioned in previous sections, and as such, we have evaluated it on top of a TBDR architecture, which inherently removes all possible overdraw. In any case, neither VRO, nor EVR nor Omega-Test are intended for reducing the geometry activity itself. In this sense, Triangle Dropping can eliminate unnecessary work at a much earlier stage than any of these previous approaches, thus exploiting its unique capability to reduce DRAM accesses to the Parameter Buffer, a feature that is completely out of the scope of such previous approaches working at the fragment level.

3 TRIANGLE DROPPING

3.1 Approach Overview

The proposed Triangle Dropping approach reduces the amount of accesses to the Parameter Buffer by eliminating the occluded geometry of a scene upfront, based on the visibility information calculated on the previous frame. Thus, Triangle Dropping allows discarding the geometry predicted as occluded early, directly in the Geometry Pipeline, rather than waiting until the middle of the Raster Pipeline, as it is the common approach for Z-Test-based techniques that work at a fragment level [1, 4, 12, 28]. One additional positive side effect of Triangle Dropping is that the *overdraw*[2] of the frame is also reduced if a conventional TBR architecture were used. However, in this work, we use a more advanced GPU as our baseline, known as **Tile-Based Deferred Rendering (TBDR)** [38], that is capable of totally eliminating overdraw by pre-computing the Z-Buffer before starting the shading of any fragment. Therefore, the benefits from Triangle Dropping reported on this article are due to savings in the Geometry Pipeline and the Tiling Engine and do not include these potential extra benefits in overdraw.

To achieve such early discard of fully occluded primitives, Triangle Dropping relies on the abundant frame-to-frame coherence present in graphics workloads. The Raster Pipeline produces very valuable information about the final visibility of a frame after rendering it; information that instead of being thrown away can be used to predict the primitives that will be visible in the next frame. However, as our approach relies on primitive-level visibility prediction, it must be very conservative when removing geometry not to introduce visible errors in the final rendered image.

Triangle Dropping operates at a primitive level by keeping track of the occluded primitives in the current frame to discard them in the next frame. Figure 6 shows a scheme of the proposed

[2]Amount of fragments that are processed by the Fragment Processors but are finally occluded.

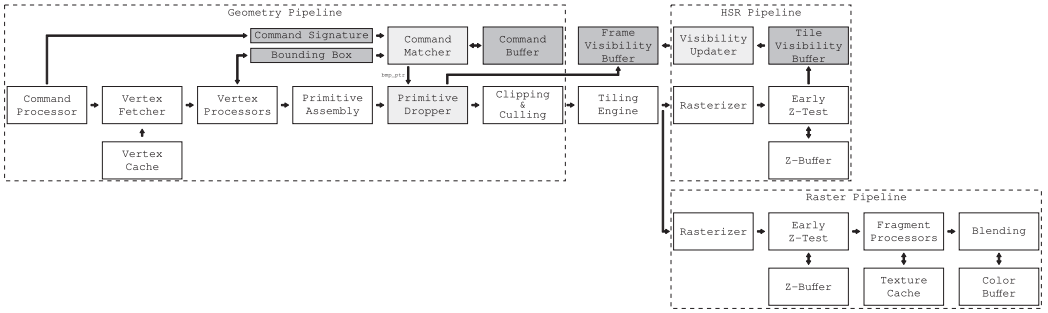

Fig. 6. Triangle Dropping implementation in a TBDR architecture. New units and memory structures appear shadowed.

Triangle Dropping approach implemented on top of a TBDR architecture. A *visibility bitmap* with one entry per primitive is associated with each command and it is used to indicate whether the primitive is visible or not. So before processing a primitive, its visibility information is checked. If the visibility bit is enabled, then we assume the primitive is visible in the current frame (as it was in the previous frame) and the primitive is processed in the Geometry Pipeline as normal. Otherwise, the primitive is dropped from the Geometry Pipeline, avoiding writes in the Parameter Buffer and any other later rasterization activity. The primitive identifier depends exclusively on the order it is issued within an OpenGL command, so it is important not to change this order across frames, otherwise, the visibility information of the primitives would be incorrect. Note, however, that the issuing order of commands (or objects) themselves does not matter. Commands could alter the order or shape of their primitives if they execute tessellations or geometry shaders. For those commands, Triangle Dropping is disabled to avoid visibility mispredictions. To do so, the graphics driver attaches a flag to the OpenGL command to indicate that it uses a geometry shader or tessellation, and this flag traverses the graphics pipeline until reaching the Triangle Dropper, which will check whether the issued command has a geometry shader attached, and if so, all of its primitives will be directly bypassed to the next stage.

3.2 Identification of Commands across Frames

Conventional GPUs do not implement a hardware-based command identifier across consecutive frames, as they are treated as totally independent entities. Obviously, at the application level, there must be command *ids* but they are not explicitly passed over to the GPU. As Triangle Dropping relies on frame-to-frame coherence, a mechanism is needed to identify commands across consecutive frames to access their *visibility bitmap* and be able to discard their occluded primitives accordingly. One simple hardware-software co-design solution could be relying on the OpenGL driver to provide these command *ids* that have been previously provided by the application itself.

However, as we cannot assume that the evaluated applications nor the OpenGL driver are providing these command *ids*, we have implemented a transparent, fully hardware-based solution for command identification across frames. Two types of information can be used for such purpose, either *static* information obtained from the OpenGL state machine (in the form of a command's signature); or *dynamic* information based on the object's bounding box (i.e., its screen coordinates in the current frame).

In the first case, a 64-bit CRC signature is computed when the Command Processor issues a command, by combining some command's *static* information (in the sense that it refers to constant properties along the command's life). In particular, we have used the following constant

parameters: the number of vertices and primitives of the command; Z-Test information (enabled bit, write mask, and depth function); blending information (enabled bit, logical blending operation function, RGB color, alpha channel, and the color mask); information from the vertex and fragment shaders (e.g., from the fragment shader, the entry point, and the number of attribute locations are considered); and finally, the primitive assembling type. The resulting 64-bit signature, combining all the aforementioned information, is stored in a register named *Command Signature*. However, the *dynamic* information used to identify commands across consecutive frames is a bounding box computed by the Vertex Processors as they process the command's vertices, which is stored in a register named *Bounding Box*.

All these tasks are managed by a Command Matcher unit that is in charge of determining whether an object (i.e., a command) in the current frame matches *itself* but in the previous frame, to use its last frame's visibility information. For that to happen, two conditions have to be met. First, both static signatures must match (bit-wise). And second, both bounding boxes must be "similar" but not necessarily exact, meaning that we allow for a *delta margin* on their screen positions as one object (or the camera) can move from one frame to the next. Further implementation details of the Command Matcher unit can be found in Section 4.1.

3.3 Visibility Checking

Figure 6 also shows the Primitive Dropper unit right after the Primitive Assembly stage. This unit accesses the command's *visibility bitmap* to check the primitive's visibility bit and then make the decision on whether to discard a given primitive or not. When an object from the current frame matches one in the Command Buffer, the *recently used bit* of the latter is set and its *bitmap pointer* is sent to the Primitive Dropper unit. A command with the *recently used bit* set cannot be matched again to avoid aliasing effects. However, if no match is found, then the current command is inserted into the Command Buffer (if there is space left), also setting its *recently used bit* to one. This avoids that two different commands in the same frame might erroneously match. A *visibility bitmap* is allocated on the Frame Visibility Buffer for the newly inserted command, with as many bits as it has primitives. The *visibility bitmap* is initialized with zeroes, indicating that all the primitives of the command are occluded, and the actual visibility of each primitive will be updated right after their rasterization.

Computing the full bounding box of a command would require that all its vertices have been processed by the Vertex Processors. However, this would incur a delay as the Primitive Dropper unit would be stalling the Geometry Pipeline while waiting to have all the vertices processed (to check for the visibility information). Furthermore, a deadlock could happen if the output queue of the Vertex Processors were full during this waiting period. To avoid such issues, only several initial vertices of each command are considered and a *partial* bounding box is calculated instead. Even though we might lose some potential by doing so, our experimental results show that using a maximum of 18 quad-vertices suffices for composing a partial bounding box, and results in a negligible loss of potential (w.r.t. a theoretical case where complete bounding boxes were used). In this way, we avoid any timing overhead.

The Frame Visibility Buffer is a global structure that stores the visibility information for all the commands in the previous frame. This table is accessed by the Primitive Dropper through a straightforward index calculation that uses the command's *bitmap pointer* (as a base address) plus the primitive *id* (recall that primitive *ids* inside a command remain the same frame-to-frame, since primitives are processed in the same order).

Regarding back-face primitives (or those that are outside the frustum view), Triangle Dropping marks them as *visible* in the Frame Visibility Buffer, as they are already discarded by the baseline TBDR GPU and not even written onto the Parameter Buffer. By tagging them as *visible*, Triangle

Dropping is ignoring those primitives that continue down the pipeline until the baseline Culling stage discards them.

For the case of primitives with the blending attribute enabled (transparent primitives), they do not update the Tile Visibility Buffer because they cannot completely hide another primitive. Given that, only opaque geometry can appear on the Tile Visibility Buffer. Whenever a transparent primitive arrives at the Primitive Dropper, it is bypassed directly to the next stage, and hence, its visibility is not determined speculatively.

3.4 Visibility Update

The final visibility of primitives is determined during the Raster Pipeline. The Tile Fetcher starts reading attributes from the Parameter Buffer, reading also the visibility pointer of each primitive, which is passed to the Rasterizer and attached to every generated quad-fragment (simply as one more attribute). In the Early Z-Test stage, we include an additional on-chip buffer named Tile Visibility Buffer. This buffer has the same dimensions as the Z-Buffer and it stores the visibility pointer of the nearest opaque fragment (the visible one so far) for each quad-fragment of the tile. The Tile Visibility Buffer is initialized with null pointers, indicating that none of the quad-fragments are visible. The visibility information of a tile is completely known when the last quad-fragment is processed by the Early Z-Test. At this point, the Visibility Updater is triggered, which iterates over non-null positions in the Tile Visibility Buffer and updates the global Frame Visibility Buffer, setting the bits of the visible primitives as needed. Once the Visibility Updater finishes, the visibility of the primitives of every command is ready to be used in the next frame. It is important to note that the Visibility Updater operates in parallel with the rest of the Raster Pipeline (i.e., the Fragment Processors and the Blending Unit) as it is implemented after the **HSR (Hidden Surface Removal)** stages, therefore, not incurring additional time penalties.

3.5 Refreshing Interval

Whenever a primitive is marked as occluded, it will be discarded upfront in the following frames, not even passing through the rest of the Geometry Pipeline. Therefore, if the primitive becomes visible in a future frame, the approach should be able to restore its *visible* status. A simple way to achieve this consists of disabling Triangle Dropping for one frame (also known as *key frame*) every certain number of frames, as a way to fully reset the visibility information. This is controlled with a *refreshing interval* parameter. Note that the shorter the refreshing interval, the fewer potential errors that might appear but also the less benefit for our technique. To provide the best result, our approach is able to dynamically adjust the refreshing interval, thus avoiding potential errors that might affect the quality of the video sequence (see Section 4.2 for implementation details).

4 HARDWARE IMPLEMENTATION DETAILS

4.1 The Command Matcher

The Command Matcher unit uses two on-chip memories: the Main Table and the Overflow Buffer. Both memories together form what we call the Command Buffer. Figure 7 shows the implementation of the Command Matcher unit along with the Command Buffer. These memories have a different number of entries that we call *sets*. A set consists of several ways (called slots) and an overflow pointer towards the Overflow Buffer. Each slot stores information about a particular command: Command Signature, Bounding Box, *valid bit*, *recently used bit*, and *bitmap pointer* (as explained in Section 3). The Main Table is indexed with the Command Signature, which requires a bit-wise exact match for a hit. Therefore, on a command lookup, if its Command Signature matches a tag, the bounding boxes of both the searched command and the one stored in the slot are

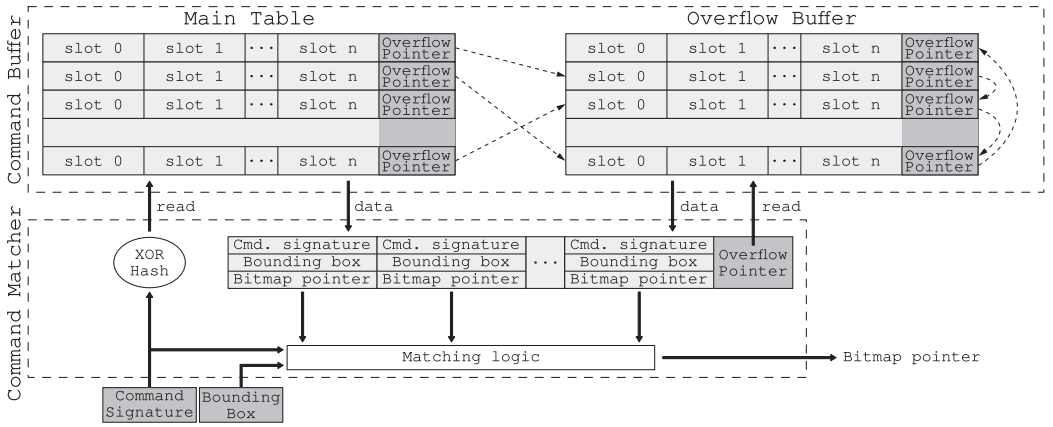

Fig. 7. Block diagram showing the Command Matcher logic and the Command Buffer structure.

compared as well. If the bounding boxes match (considering the delta margin), then the *bitmap pointer* stored in the matched slot is retrieved and driven to the Primitive Dropper (the bounding box is also updated with the one from the input). Otherwise, if no matching is found in the Main Table, the Overflow Buffer is searched. If the searched command is neither found in the Overflow Buffer, then the command is inserted into the Command Buffer.

The insertion of a command into the Command Buffer works as follows. First, it is checked whether there is a free slot in the current set. If so, then both the Command Signature and the Bounding Box inputs are written into that slot, setting the *valid* and *recently used* bits to one. Whenever a new command is inserted into the Command Buffer, a new *visibility bitmap* is allocated for it (with as many bits as input primitives—provided if there is space available in the Frame Visibility Buffer) and the *bitmap pointer* is written into the slot. Otherwise, if all the slots in the set are occupied, the *overflow pointer* is used to access the Overflow Buffer where the search of a free slot continues. If the set in the Overflow Buffer is also full, then the ending *overflow pointer* will point to another set, in a linked-list fashion (as illustrated in Figure 7). The search continues until a free slot is found. Alternatively, if a set is full but the *overflow pointer* has a null value, a new free set is allocated and chained. Finally, if the entire Overflow Buffer is full, the insertion cannot be done and a null pointer is driven to the Primitive Dropper unit. This indicates that the command was not found in the Command Buffer and, therefore, it is treated as a conventional visible command, resulting in some potential loss for Triangle Dropping. At the end of a frame, the *recently used* bits are cleared, and the commands that have not been used are deleted from the Command Buffer to free up space.

To properly dimension the Command Buffer, different experiments have been performed. Out of the evaluated benchmarks, the one with more commands is *bbr*, with up to 312 commands in a frame. That means a size of 12.56 KiB for each table in the Command Buffer. Similarly, to estimate the storage needs of the Frame Visibility Buffer, the maximum number of primitives in a frame were counted. This number was 190,449 primitives in a single frame, which translates into a Frame Visibility Buffer of about 23.25 KiB. Such sizes are very reasonable in a modern GPU. Therefore, for our evaluation, we have rounded up those sizes and used 16+16 KiB for the Command Buffer and 32 KiB for the Frame Visibility Buffer. Note, however, that if a benchmark exceeds these sizes in a particular frame, then the only downside for Triangle Dropping is a loss of potential, that according to our experimental results when smaller sizes were considered, it is almost negligible.

4.2 Handling Intermittent Primitives

There are some commands that, due to the nature of their movements (and/or the environment around them), have primitives whose visibility varies over a range of frames. It might be the case of a rotating object where triangles that are partially occluded by another object may become visible after a few frames because of their rotating movement (e.g., a rotating wheel that is also partially covered by the chassis of the car). Once Triangle Dropping marks such kind of intermittent primitives as invisible, it will predict a wrong visibility if they later appear, leading to an error in the image. This visibility artifact could be more noticeable when a *key frame* arrives, since the correct visibility is computed and wrongly discarded primitives would become suddenly visible. We refer to primitives that change their visibility over time as *intermittent primitives*.

To overcome the potential issues of such intermittent primitives and not to cause perceivable errors, they must be detected so Triangle Dropping can ignore them. To achieve that, we use a heuristic that consists of detecting whether the visibility of a primitive changes from invisible to visible across consecutive frames. Since Triangle Dropping does not update the real visibility until a key frame arrives, this task is performed by the Visibility Updater whenever a key frame is processed. Before overwriting the visibility status of a primitive in the Frame Visibility Buffer, its previous visibility is checked. If a change from invisible to visible is found, then the primitive is conservatively annotated as intermittent. Such property is tracked in the visibility bitmap with two extra bits per entry: *intermittent bit* and *previous visibility bit*. Therefore, when querying the visibility of a primitive, the Primitive Dropper checks its intermittent bit. If set, then the primitive is ignored and treated as any other conventional primitive by the GPU, regardless of its last frame's visibility status.

Dynamic refreshing interval. Intermittent primitives need a warming-up period before they can be marked as intermittent. We introduce a mitigation mechanism aimed at minimizing this warming-up period by using a *dynamic refreshing interval* to capture an intermittent pattern in an early manner. The approach starts with the shortest refreshing interval of 2 (i.e., one *key frame* every 2 frames). At each key frame, it is determined if new commands have entered the scene since the previous key frame. If there are no new commands, then the refreshing interval is increased. Otherwise, it is reset to the minimum value of 2. The maximum value for the refreshing interval has been determined experimentally. In particular, we measured that refreshing every 5 frames results in a negligible loss of potential for Triangle Dropping while keeping a close-to-perfect image quality (refer to Section 5.3). Although this dynamic approach could result in a loss of potential, in the long run, the effect is negligible, as commands tend to stay in the scene for a high number of frames, thus leveraging its intermittent visibility information.

4.3 Limitations of Triangle Dropping

Although Triangle Dropping efficiently solves the Hidden Surface Removal problem, it still has some limitations with respect to image quality due to its predictive nature. As described above, the so-called intermittent primitives may lead to wrong visibility predictions if they become occluded in a given frame (and so they are tagged as invisible by Triangle Dropping) but they become visible after several frames. In such cases, our technique will generate mispredictions for such intermittent primitives until the next *key frame* arrives and the correct visibility status is calculated.

To overcome this issue and to avoid perceivable image artifacts by the user, Triangle Dropping incorporates two mitigating measures, as explained in Section 4.2. First, a mechanism to detect intermittent primitives is included so once such a primitive is detected, it will be marked as intermittent and it will not be considered for being dropped anymore. And, second, a dynamic refreshing interval is also implemented to minimize the time needed to detect intermittent primitives.

In particular, the mechanism starts setting *key frames* every other frame to quickly capture intermittent patterns. It is worth noting that at a conventional frame rate of 60 frames/second, just 33 milliseconds are needed to detect an intermittent primitive. This short period plus the fact that intermittent primitives are not very common result in not perceivable image artifacts by the human visual system, as it is analyzed and quantified in Section 5.3 by using the MSSIM and VSSIM perceptually based quality metrics that calculate the similarity between two images or video sequences, respectively. Please, refer to Section 5.3 for further details on how image quality is affected by using Triangle Dropping.

5 EVALUATION METHODOLOGY

5.1 Simulator Infrastructure

To evaluate the performance and energy improvement achieved with the proposed Triangle Dropping approach, we use Teapot [7], a simulation framework that, among other things, models a Mali T-450 GPU [6], a TBR architecture widely used in mobile devices. Teapot also models a TBDR architecture [38], which is the baseline GPU architecture we have used for our evaluations, since it completely removes all overdraw. This way, we can isolate the benefits of Triangle Dropping in the Geometry Pipeline and the Tiling Engine and report a lower bound of its improvements, since in this case Triangle Dropping does not benefit from a potential reduction in overdraw. Power models are integrated from other widely used tools: McPAT [36], a tool that calculates the power, area, and timing of digital designs; and DRAMSim2 [49] for modeling DRAM and the memory controllers.

Simulator traces have been obtained with GAPID [23], a tool-set for debugging OpenGL [54] graphics on Android devices. With the `gapii` tool, we intercept all the OpenGL calls issued by applications and store them in a trace file. Either a real smartphone or an **Android Virtual Device (AVD)** [22] can be used for this purpose. Due to the high overhead of intercepting OpenGL calls, it is difficult to obtain trace files whose frames flow sufficiently smoothly to exhibit abundant frame-to-frame coherence. To overcome this infrastructure issue, we have instrumented `gapii` to obtain a modified intercepting mechanism for functions that return the internal clock status of the device (`clock_gettime` or `gettimeofday`) and provide a slower clock tick. With the `gapir` tool, we can replay this trace on top of an instrumented Gallium Softpipe Driver [19] to generate a trace file readable by Teapot. This trace contains information about the GPU pipeline execution, such as vertices, primitives, shader programs or fragments, and it is consumed by the cycle-accurate simulator.

Table 1 shows the simulation parameters used to evaluate our benchmarks, as well as the configuration of the logic units and memory structures introduced by Triangle Dropping. The area, power, and timing of all the new structures have been modeled in the simulator. In particular, the area overhead introduced by Triangle Dropping has been measured to be 1.1% of the total die, of which 0.54% corresponds to the Command Buffer, 0.40% corresponds to the Frame Visibility Buffer, and 0.16% to the Tile Visibility Buffer. With respect to the dissipated power, Triangle Dropping incurs a 0.84% power overhead, of which 0.34% corresponds to the Main Table and the Overflow Buffer in almost equal parts, 0.31% comes from the Frame Visibility Buffer, and the remaining 0.19% corresponds to the Tile Visibility Buffer.

5.2 Benchmarks

To evaluate Triangle Dropping, we have used a set of very popular games (based on their number of downloads) from the Google Play Store [24]. The evaluated scenes have been carefully selected to get a representative, common, and realistic use-case scenario for each game. Table 2 enumerates the evaluated games along with some specific characteristics of each working set.

Table 1. Simulation Parameters

Baseline GPU Parameters	
Frequency	600 MHz
Voltage	1.0 V
Technology node	22 nm
Screen Resolution	2,160 × 1,080
Tile Size	16 × 16 pixels
Main Memory	
Frequency	400 MHz
Voltage	1.5 V
Technology node	32 nm
Latency	50–100 cycles
Bandwidth	4 B/cycle (dual channel LPDDR3)
Line Size	64 bytes
Size	1 GiB, 8 banks
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 (×4)	8 KiB, 1 bank, 2 cycles
Tile Cache	32 KiB, 1 bank, 2 cycles
L2 Cache	256 KiB, 8 banks, 18 cycles
Color Buffer	1 KiB, 1 bank, 1 cycle
Depth Buffer	1 KiB, 1 bank, 1 cycle
Non-programmable stages	
Primitive assembly	1 vertex/cycle
Rasterizer	1 attribute/cycle
Early Z-Test	8 in-flight quad-fragments
Programmable stages	
Vertex Processor	4 vertex processors
Fragment Processor	4 fragment processors
Triangle Dropping hardware	
Main Table	16 KiB (32 lines, 16-way associative)
Overflow Buffer	16 KiB (32 lines, 16-way associative)
Frame Visibility Buffer	32 KiB
Tile Visibility Buffer	1 KiB, 1 bank, 1 cycle
Primitive Dropper	1 primitive/cycle
Visibility Updater	2 pointers/cycle, 8 in-flight pointers

Table 2. Evaluated Benchmarks from the Google Play Store

Benchmark	Alias	Description	Downloads (Mill.)	Vertex shader instr. (Mill.)	Fragment shader instr. (Mill.)	Execution Time (Mill. cycles)
Beach Buggy Racing	bbr	Racing	100–500	96	2,052	749
Derby Destruction Simulator	dds	Racing & Battle Royale	10–50	165	4,993	1,140
Gravity	gra	Action	1–5	74	355	144
Hellrider	hrd	Racing	1–5	112	3,534	868
Hot Wheels	hwl	Racing	50–100	431	2,073	950
Maze 3D	maz	Labyrinth	10–50	131	4,420	1,112
Sniper 3D	s3d	Shooter	100–500	144	1,600	684
Sonic Dash	snd	Adventure arcade	100–500	87	4,154	1,219

5.3 Image Quality

Image quality has been quantified by comparing a frame when Triangle Dropping is enabled with the same frame from the baseline GPU by using the **MSSIM (Mean Structural Similarity Index Measure)** metric [56], a widely adopted, perceptually based quality metric that computes the

Table 3. Image and Video Quality of the Evaluated Benchmarks Using Both MSSIM and VSSIM Metrics

Benchmark	min. MSSIM	avg. MSSIM	min. VSSIM	VSSIM
bbr	0.99925	0.999944	0.999829	0.999849
dds	0.999447	0.999957	0.999869	0.999919
gra	0.99981	0.999951	0.999593	0.999782
hrd	0.999161	0.999852	0.999618	0.99969
hwl	0.995626	0.999253	0.99863	0.998913
maz	0.999371	0.999979	0.99989	0.999951
s3d	0.992689	0.99894	0.991632	0.995344
snd	0.999849	0.999991	0.999919	0.999937

similarity between two images. The MSSIM performs better than other similarity metrics that just measure differences in pixel color (such as PSNR or MSE), as it correlates better with the perception of the human visual system [21]. The MSSIM index is a number in the [0, 1] range, where a 1 means an exact matching, and the human-perceptible threshold is 0.95 [18]. In a later work, the same authors extended MSSIM to VSSIM [57], another structural similarity index aimed at evaluating the quality of video sequences.

To perform a more insightful study of the resulting image quality when Triangle Dropping is applied, we have used both MSSIM and VSSIM metrics. For each benchmark, Table 3 shows the minimum and the average MSSIM; and also the minimum and final VSSIM of all its frames. As it can be seen, all of the measured MSSIM values are not only above the 0.95 threshold but the *minimum* MSSIM value for each benchmark is always above 0.99 with an average MSSIM of about 1 on each game. Furthermore, when considering the full sequence of frames, the minimum video quality achieved according to the VSSIM metric is always above 0.99 on all the evaluated games. Therefore, we can conclude that our technique does not incur any perceivable error.

6 EXPERIMENTAL RESULTS

6.1 Geometry Reduction

Our first set of experiments quantifies the amount of scene geometry that Triangle Dropping is capable of removing. Figure 8 shows a breakdown of the primitives that are finally written into the Parameter Buffer. Comparing the baseline ("base" on the x axis) with our approach ("td" on the x axis), it can be seen that Triangle Dropping can remove an average of 31.38% of the primitives that are written into the Parameter Buffer. Note that this represents a large fraction of the total number of the *opaque* occluded primitives (56.99%).

6.2 Memory Bandwidth Reduction

Since the major goal of Triangle Dropping is reducing accesses to the Parameter Buffer in DRAM, an interesting parameter to analyze is the impact that Triangle Dropping has on the memory traffic. Figure 9 shows the achieved DRAM traffic reduction. The bars differentiate accesses to the Parameter Buffer (red stack) from *other* accesses (blue stack). We observe that the reduced fraction comes from Parameter Buffer accesses, as expected, except for *hrd* and *snd*, where the "other" DRAM accesses are also reduced (they actually correspond to memory accesses for fetching textures, since the L2 is not polluted by occluded geometry and, hence, more capacity is available for other data types). Overall, Triangle Dropping achieves an average memory traffic reduction of 16.92%.

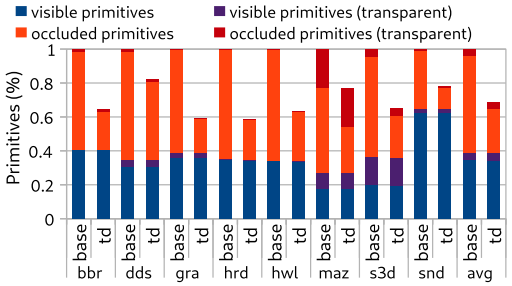

Fig. 8. Geometry reduction achieved by Triangle Dropping.

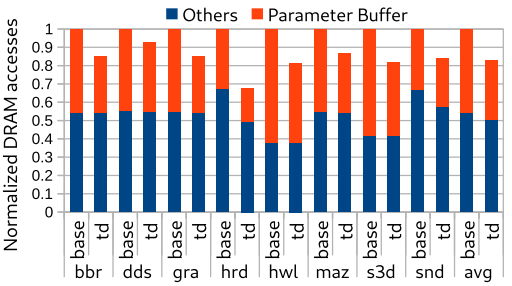

Fig. 9. DRAM traffic reduction when using Triangle Dropping.

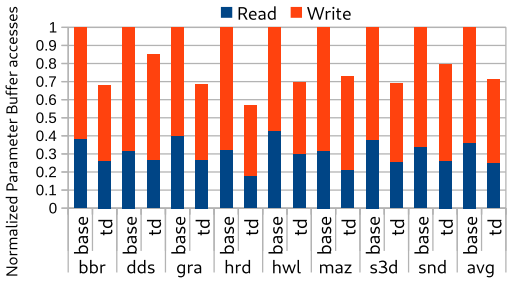

Fig. 10. Parameter Buffer accesses that end up going to DRAM.

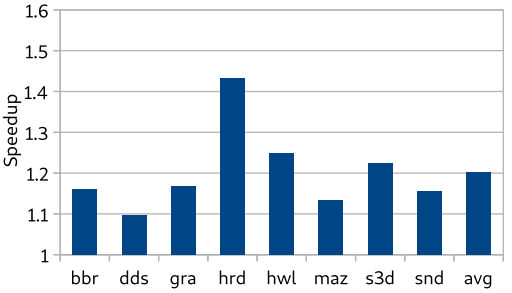

Fig. 11. Speedup achieved by Triangle Dropping over a TBDR architecture.

For additional details, Figure 10 shows the Parameter Buffer accesses (distinguishing between reads and writes) that finally reach DRAM. On average, Triangle Dropping is capable of reducing 28.78% of those accesses (10.92% from reads and 17.86% from writes). Note that these accesses correspond to primitives that are inside the camera frustum volume and are not back-face culled.

6.3 Performance and Energy Efficiency

Because of the dropped primitives, many computations are avoided in addition to the saved memory accesses, resulting in a positive impact on performance. Figure 11 shows the speedup achieved by Triangle Dropping over a TBDR architecture, obtaining an average speedup of 20.24%. In the particular case of *hrd*, which achieves a speedup of 43.17%, the improvement not only comes from eliminated Parameter Buffer accesses but also from texture accesses that end up going to DRAM. The latter happens because the dropped geometry leads to more space in the shared L2 to allocate texture blocks, thus reducing the DRAM traffic coming from textures.

Figure 12 shows the global energy savings achieved when applying Triangle Dropping on top of a TBDR GPU. Since the memory accesses are reduced, the energy consumed by the memory system is reduced accordingly. On average, Triangle Dropping achieves 14.5% energy savings, of which 11.7% comes from the memory system and 2.8% comes from the GPU activity. The energy savings in the GPU are mainly due to the activity reduction in the Clipping&Culling stage, the Polygon List Builder, and the Tile Fetcher. These energy savings can be as high as 21.59% for *hrd*. As expected, the overall energy savings of Triangle Dropping correlate well with the memory bandwidth reductions reported in Figure 9.

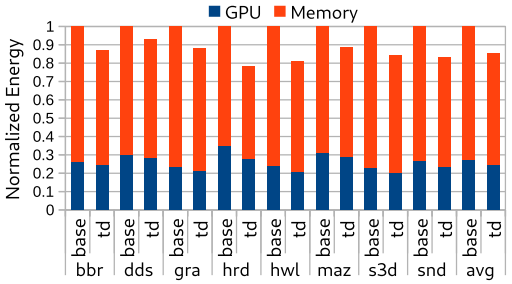

Fig. 12. Energy savings of Triangle Dropping over a TBDR architecture.

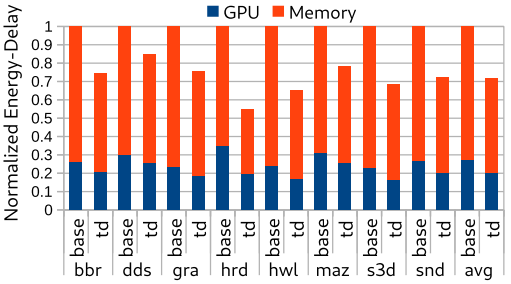

Fig. 13. Energy-Delay Product (EDP) savings of Triangle Dropping over a TBDR architecture.

Regarding the combined energy-performance efficiency of Triangle Dropping, we also report the **Energy-Delay Product (EDP)** in Figure 13. On average, our approach achieves EDP savings of 28.23% and up to 45.23% in the case of *hrd*.

As mentioned earlier, it is worth noting that Triangle Dropping does not eliminate shading/rendering activities for eventually occluded fragments (an effect known as overdraw), since we have implemented it on top of a TBDR architecture, which totally eliminates fragment overdraw on a scene, thanks to the Hidden Surface Removal stage included in the Raster Pipeline that operates at a fragment level. Differently, Triangle Dropping operates much earlier, in the Geometry Pipeline, at a primitive level.

7 CONCLUSIONS

Mobile device game users demand increasingly more complex scenes, with a lot of occluded geometry that results in a huge waste of GPU and memory resources. To overcome that, we have proposed Triangle Dropping, a novel micro-architecture approach for mobile GPUs that drastically reduces the occluded geometry in a scene by leveraging the visibility of the primitives from the previous frame. We show that this technique removes 31.38% of the primitives for a set of representative benchmarks. We have implemented our approach on top of a TBDR GPU pipeline, since its major goal is to reduce the activity in the Geometry Pipeline and in the Tiling Engine, which cannot be eliminated by the Hidden Surface Removal stage. Our experimental results for a set of popular games show that Triangle Dropping achieves an average speedup of 20.2% in addition to average energy savings of 14.5% and an average **energy-delay product (EDP)** reduction of 28.2%.

REFERENCES

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

[2] Tomas Akenine-Moller and Jacob Strom. 2008. Graphics processing units for handhelds. *Proc. IEEE* 96, 5 (2008), 779–789.

[3] Maria-Elena Algorri and Francis Schmitt. 1996. Mesh simplification. In *Computer Graphics Forum*, Vol. 15. Wiley Online Library, 77–86.

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

[5] Iosif Antochi, Ben Juurlink, Stamatis Vassiliadis, and Petri Liuha. 2004. Memory bandwidth requirements of tile-based rendering. In *Proceedings of the International Workshop on Embedded Computer Systems*. Springer, 323–332.

[6] Arm. 2021. ARM Mali-450 GPU. (2021). Retrieved from https://developer.arm.com/products/graphics-and-multimedia/mali-gpus/mali-450-gpu.

[7] Jose-Maria Arnau, Joan-Manuel Parcerisa, and Polychronis Xekalakis. 2013. 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, 37–46.

[8] Richard Broadhurst, John Howson, and Robert Theed. 2019. Using tiling depth information in hidden surface removal in a graphics processing system. (Dec. 17, 2019). US Patent 10,510,182.

[9] Paolo Cignoni, Claudio Montani, and Roberto Scopigno. 1998. A comparison of mesh simplification algorithms. *Comput. Graph.* 22, 1 (1998), 37–54.

[10] David Corbalan-Navarro, Juan L. Aragón, Marti Anglada, Enrique De Lucas, Joan-Manuel Parcerisa, and Antonio Gonzalez. 2021. Omega-Test: A predictive early-Z culling to improve the graphics pipeline energy-efficiency. *IEEE Trans. Visualiz. Comput. Graph.* https://doi.org/10.1145/3527861

[11] Microsoft Corporation. 2003. *Microsoft DirectX 9 Programmable Graphics Pipeline.* Microsoft Press.

[12] Enrique De Lucas, Pedro Marcuello, Joan-Manuel Parcerisa, and Antonio González. 2018. Visibility rendering order: Improving energy efficiency on mobile GPUs through frame coherence. *IEEE Trans. Parallel Distrib. Syst.* 30, 2 (2018), 473–485.

[13] Christopher DeCoro and Natalya Tatarchuk. 2007. Real-time mesh simplification using the GPU. In *Proceedings of the Symposium on Interactive 3D Graphics and Games.* 161–166.

[14] Angus Dorbie. 2005. Method and apparatus for early culling of occluded objects. (Aug. 2, 2005). US Patent 6,924,801.

[15] Jihad El-Sana, Neta Sokolovsky, and Cláudio T. Silva. 2001. Integrating occlusion culling with view-dependent rendering. In *Proceedings of the Conference on Visualization (VIS'01).* IEEE, 371–575.

[16] Andreas Due Engh-Halstvedt, Alexander Eugene Chalfin, and Frode Heggelund. 2021. Hidden surface removal in graphics processing systems. (June 8, 2021). US Patent 11,030,783.

[17] Robert Farrell. 2009. Graphics processing with hidden surface removal. (Dec. 31, 2009). US Patent App. 12/215,920.

[18] Jeremy R. Flynn, Steve Ward, Julian Abich, and David Poole. 2013. Image quality assessment using the SSIM and the just noticeable difference paradigm. In *Proceedings of the International Conference on Engineering Psychology and Cognitive Ergonomics.* Springer, 23–30.

[19] Freedesktop. 2021. Gallium3D. Retrieved from https://www.freedesktop.org/wiki/Software/gallium.

[20] Yaosheng Fu, Evgeny Bolotin, Niladrish Chatterjee, David Nellans, and Stephen W. Keckler. 2021. GPU domain specialization via composable on-package architecture. *ACM Trans. Archit. Code Optim.* 19, 1 (Dec. 2021).

[21] Xinbo Gao, Wen Lu, Dacheng Tao, and Xuelong Li. 2009. Image quality assessment based on multiscale geometric analysis. *IEEE Trans. Image Process.* 18, 7 (2009), 1409–1423.

[22] Google. 2021. Android SDK. Retrieved from https://developer.android.com/studio.

[23] Google. 2021. GAPID. Retrieved from https://developers.google.com/vr/develop/unity/gapid.

[24] Google. 2021. Google Play. Retrieved from https://play.google.com.

[25] Nilanjan Goswami, Derek Lentz, Adithya Hrudhayan Krishnamurthy, and David C. Tannenbaum. 2021. Efficient redundant coverage discard mechanism to reduce pixel shader work in a tile-based graphics rendering pipeline. (May 18, 2021). US Patent 11,010,954.

[26] Edward Colton Greene and Patrick Matthew Hanrahan. 2002. Method and apparatus for occlusion culling in graphics systems. (Nov. 12, 2002). US Patent 6,480,205.

[27] Edward C. Greene, Douglas A. Voorhies, Paolo Sabella, John M. Danskin, and James M. Van Dyke. 2005. Occlusion culling method and apparatus for graphics systems. (May 17, 2005). US Patent 6,894,689.

[28] Ned Greene, Michael Kass, and Gavin Miller. 1993. Hierarchical Z-buffer visibility. In *Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques.* ACM, 231–238.

[29] Bernd Hamann. 1994. A data reduction scheme for triangulated surfaces. *Comput.-aid. Geom. Des.* 11, 2 (1994), 197–214.

[30] Jon Hjelmervik and Jean-Claude Léon. 2007. GPU-accelerated shape simplification for mechanical-based applications. In *Proceedings of the IEEE International Conference on Shape Modeling and Applications (SMI'07).* IEEE, 91–102.

[31] Hugues Hoppe, Tony DeRose, Tom Duchamp, John McDonald, and Werner Stuetzle. 1993. Mesh optimization. In *Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques.* 19–26.

[32] Liang Hu, Xilong Che, and Si-Qing Zheng. 2016. A closer look at GPGPU. *ACM Comput. Surv.* 48, 4 (Mar. 2016).

[33] Harold Hubschman et al. 1982. Frame-to-frame coherence and the hidden surface computation: Constraints for a convex world. *ACM Trans. Graph.* 1, 2 (1982), 129–162.

[34] Tom Hudson, Dinesh Manocha, Jonathan Cohen, Ming Lin, Kenneth Hoff, and Hansong Zhang. 1997. Accelerated occlusion culling using Shadow Frusta. In *Proceedings of the 13th Annual Symposium on Computational Geometry.* 1–10.

[35] Brian Jacobson. 2011. Method for accelerated determination of occlusion between polygons. (Mar. 8, 2011). US Patent 7,903,108.

[36] Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. 2009. 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, 469–480.

[37] Jieun Lim, Nagesh B. Lakshminarayana, Hyesoon Kim, William Song, Sudhakar Yalamanchili, and Wonyong Sung. 2014. Power modeling for GPU architectures using McPAT. *ACM Trans. Des. Autom. Electron. Syst.* 19, 3 (2014), 26.

[38] Imagination Technologies Limited.. PowerVR Hardware. Architecture Overview for Developers. Retrieved 23 Nov 2018 from http://cdn.imgtec.com/sdk-documentation/PowerVR+Hardware.Architecture+Overview+for+Developers.pdf.

[39] Peter Lindstrom. 2000. Out-of-core simplification of large polygonal models. In *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*. 259–262.

[40] Jun Liu, Wei Ding, Ohyoung Jang, and Mahmut Kandemir. 2013. Data layout optimization for GPGPU architectures. *SIGPLAN Not.* 48, 8 (Feb. 2013), 283–284.

[41] Haik Lorenz and Jürgen Döllner. 2008. Dynamic mesh refinement on GPU using geometry shaders. In *Proceedings of the 16th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision*.

[42] Yashuai Lü, Hui Guo, Libo Huang, Qi Yu, Li Shen, Nong Xiao, and Zhiying Wang. 2021. GraphPEG: Accelerating graph processing on GPUs. *ACM Trans. Archit. Code Optim.* 18, 3 (May 2021).

[43] David Luebke, Mark Harris, Naga Govindaraju, Aaron Lefohn, Mike Houston, John Owens, Mark Segal, Matthew Papakipos, and Ian Buck. 2006. GPGPU: General-purpose computation on graphics hardware. In *Proceedings of the ACM/IEEE Conference on Supercomputing*. 208–es.

[44] David P. Luebke. 2001. A developer's survey of polygonal simplification algorithms. *IEEE Comput. Graph. Applic.* 21, 3 (2001), 24–35.

[45] Alexandros Papageorgiou and Nikos Platis. 2015. Triangular mesh simplification on the GPU. *Vis. Comput.* 31, 2 (2015), 235–244.

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

[47] Chao Peng and Yong Cao. 2012. A GPU-based approach for massive model rendering with frame-to-frame coherence. In *Computer Graphics Forum*, Vol. 31. Wiley Online Library, 393–402.

[48] Jeff Pool. 2012. *Energy-precision Tradeoffs in the Graphics Pipeline*. Ph.D. Dissertation. The University of North Carolina at Chapel Hill.

[49] Paul Rosenfeld, Elliott Cooper-Balis, and Bruce Jacob. 2011. DRAMSim2: A cycle accurate memory system simulator. *IEEE Comput. Archit. Lett.* 10, 1 (2011), 16–19.

[50] Jarek Rossignac and Paul Borrel. 1993. Multi-resolution 3D approximations for rendering complex scenes. In *Modeling in Computer Graphics*. Springer, 455–465.

[51] William J. Schroeder, Jonathan A. Zarge, and William E. Lorensen. 1992. Decimation of triangle meshes. In *Proceedings of the 19th Annual Conference on Computer Graphics and Interactive Techniques*. 65–70.

[52] Graham Sellers and John Kessenich. 2016. *Vulkan Programming Guide: The Official Guide to Learning Vulkan*. Addison-Wesley Professional.

[53] M. Shebanow. 2013. An evolution of mobile graphics. *Keynote Talk at High Performance Graphics* (2013). https://www.highperformancegraphics.org/wp-content/uploads/2013/Shebanow-Keynote.pdf.

[54] Dave Shreiner, The Khronos OpenGL ARB Working Group. 2009. OpenGL Programming Guide: The Official Guide to Learning OpenGL, Versions 3.0 and 3.1. Pearson Education 719 https://www.highperformancegraphics.org/wp-content/uploads/2013/Shebanow-Keynote.pdf.

[55] Pengyu Wang, Jing Wang, Chao Li, Jianzong Wang, Haojin Zhu, and Minyi Guo. 2021. GRUs: Toward unified-memory-efficient high-performance graph processing on GPU. *ACM Trans. Archit. Code Optim.* 18, 2 (Feb. 2021).

[56] Zhou Wang, Alan C. Bovik, Hamid R. Sheikh, and Eero P. Simoncelli. 2004. Image quality assessment: from error visibility to structural similarity. *IEEE Trans. Image Process.* 13, 4 (2004), 600–612.

[57] Zhou Wang, Ligang Lu, and Alan C. Bovik. 2004. Video quality assessment based on structural distortion measurement. *Sig. Process.: Image Commun.* 19, 2 (2004), 121–132.

[58] Andrew Wilson, Ketan Mayer-Patel, and Dinesh Manocha. 2001. Spatially-encoded far-field representations for interactive walkthroughs. In *Proceedings of the Ninth ACM International Conference on Multimedia*. ACM, 348–357.

[59] Michael Wimmer and Jiří Bittner. 2005. Hardware occlusion queries made useful. In *GPU Gems 2: Programming Techniques for High-performance Graphics and General-purpose Computation*. Addison-Wesley.

[60] Yu Zhang, Da Peng, Xiaofei Liao, Hai Jin, Haikun Liu, Lin Gu, and Bingsheng He. 2021. LargeGraph: An efficient dependency-aware GPU-accelerated large-scale graph processing. *ACM Trans. Archit. Code Optim.* 18, 4 (Sep. 2021).