Early Visibility Resolution for Removing Ineffectual Computations in the Graphics Pipeline

Martí Anglada[1] Enrique de Lucas[2] Joan-Manuel Parcerisa[1] Juan L. Aragón[3] Antonio González[1]
[1]Universitat Politècnica de Catalunya [2]Semidynamics Technology Services, Barcelona [3]Universidad de Murcia
{manglada, jmanel, antonio}@ac.upc.edu
enrique.delucas@semidynamics.com jlaragon@um.es

***Abstract*—GPUs' main workload is real-time image rendering. These applications take a description of a (animated) scene and produce the corresponding image(s). An image is rendered by computing the colors of all its pixels. It is normal that multiple objects overlap at each pixel. Consequently, a significant amount of processing is devoted to objects that will not be visible in the final image, in spite of the widespread use of the Early Depth Test in modern GPUs, which attempts to discard computations related to occluded objects.**

Since animations are created by a sequence of similar images, visibility usually does not change much across consecutive frames. Based on this observation, we present Early Visibility Resolution (EVR), a mechanism that leverages the visibility information obtained in a frame to predict the visibility in the following one. Our proposal speculatively determines visibility much earlier in the pipeline than the Early Depth Test. We leverage this early visibility estimation to remove ineffectual computations at two different granularities: pixel-level and tile-level. Results show that such optimizations lead to 39% performance improvement and 43% energy savings for a set of commercial Android graphics applications running on state-of-the-art mobile GPUs.

***Keywords*-Graphics Pipeline; Tile-Based Rendering; Energy Efficiency; Visibility;**

I. Introduction

Graphics-rich applications are overwhelmingly common in mobile platforms. However, achieving an acceptable frame rate in such applications often requires a significant amount of energy, a very constrained resource in battery-operated devices.

Frames are rendered by processing data through the graphics pipeline: vertices are transformed, lighted and grouped into primitives, which are usually triangles. Triangles are then rasterized into fragments, pixel-sized regions of primitives, upon which a final color is computed and written to the framebuffer in main memory to be displayed. The main cause of energy consumption in the described pipeline is communication with main memory, a great fraction of which is devoted to fragment operations: colors are typically computed after accessing stored images named textures and the resulting color has to be stored in the framebuffer. Tile-Based Rendering (TBR) is a rendering paradigm widely used in mobile devices to reduce memory accesses, because it divides the screen into rectangular sections or *tiles*, that are independently rendered over small on-chip buffers.

This paper presents Early Visibility Resolution (EVR), a novel hardware mechanism implemented on a TBR architecture that improves the energy efficiency of GPUs by enabling optimizations on existing techniques such as Early-Z Test and Rendering Elimination.

Improving the Early-Z test: Visibility of overlapping fragments is typically handled employing the Z Buffer, a memory region which stores the depth of the closest fragment to the camera for every pixel in the frame. New fragments compare their depth with the one stored in their same position and are only written to the framebuffer if they are closer than the previously visible fragment. However, it is common for the color of pixels to be computed multiple times, a phenomenon known as *overshading*, as values produced by previously-computed fragments are occluded by newer fragments if they turn to be closer to the observer. To reduce overshading, most GPUs nowadays include an additional Early-Z test just before the stage that computes the fragment color. However the requirement for the Early-Z test to reduce the overshading is that opaque primitives are processed in front-to-back order so that hidden primitives are processed after visible ones. Our proposed EVR technique reduces the overshading caused by hidden primitives by identifying them well before they are rasterized, and scheduling them after the visible ones, thus ensuring that they will be rejected by the Early-Z test.

Improving Rendering Elimination (RE): RE [1] is a technique implemented on top of a TBR architecture that identifies tiles that do not change between two consecutive frames. RE keeps track of the primitives in each tile and the input data used for computing their colors. Before starting the rendering process of a tile, its input data is compared against that used to compute the same tile in the previous frame and, if they match, the rendering of the entire tile is skipped because its colors, stored in the framebuffer, will not change. However, we observe that a significant potential is lost because many equal tiles cannot be identified as such when the only changes between frames occur in hidden primitives that do not contribute to the final colors of the tiles. Since EVR identifies hidden primitives early, they can

2378-203X/19/$31.00 ©2019 IEEE
DOI 10.1109/HPCA.2019.00015

IEEE computer society

be excluded from the tile input data set that RE compares, thus increasing the amount of skipped tiles.

Early Visibility Resolution estimates visibility in early stages of the graphics pipeline by exploiting frame-to-frame coherence. Given that animations are produced as a sequence of similar images, visibility tends to remain very similar across consecutive frames: occluded primitives in a frame are prone to be occluded in the following frame as well. The visibility of a primitive is separately determined for every tile it overlaps in order to achieve a better precision, since it may occur that a primitive is partially visible, i.e., visible in some tiles but completely occluded in some other tiles. Our proposal is based on computing the depth of the farthest visible point of each tile in a frame and use these depths to predict the visibility of primitives in the next frame. Whenever a primitive is assigned to a tile, its closest point to the camera is compared against the depth of the farthest visible point for that tile in the previous frame: if the former is farther, the primitive is considered to be occluded for that tile, whereas if it is closer, the primitive is considered to be visible. We leverage this early visibility prediction scheme to reduce redundancy in a TBR Graphics Pipeline at two different granularities:

- Fragment level. The effectiveness of the traditional Early-Z test hidden fragment rejection is improved by processing primitives predicted to be occluded after those predicted to be visible.
- Tile level. The effectiveness of Rendering Elimination's redundant tile detection is significantly improved by ignoring primitives predicted to be occluded when computing similarities between tiles.

In the Results section, we show that early detection and reordering of potentially occluded primitives reduces overshading by 20%, and boosts redundant tile detection by 5%, yielding speedups of 39% and energy reduction of 43%

II. Background

Tile-Based Rendering. The Graphics Pipeline consists of a sequence of steps required to render a scene generated by an application. Figure 1 shows the block diagram of a hardware implementation of such pipeline based on the architecture of an ARM Mali-450 [2], one of the most widespread GPU architectures on the mobile market [3]. The pipeline execution is initiated by the application, which sends API commands to the GPU to be processed. The Command Processor receives these commands and sets the appropriate state for the Graphics Pipeline. Among them, streams of vertices are sent as *draw commands*, to be processed by the pipeline. In Tile-Based Rendering (TBR) the rendering process is divided into two pipelines: Geometry and Raster. Furthermore, the screen space is partitioned into regular, independently-rendered regions named *tiles*, which allows for the use of smaller on-chip memories for storage of temporary depth and color values. The Geometry Pipeline fetches vertices and their *attributes* (per-vertex information such as position, color or texture coordinates) from main memory and transforms the vertices from object-space coordinates into the display-space coordinates using user-defined programs called *vertex shaders* and some fixed function stages of the pipeline. The vertices are grouped into primitives (usually triangles) in the Primitive Assembly stage, which also discards the ones that do not face the camera or are outside the viewing volume. The Geometry Pipeline finishes its process in the Polygon List Builder stage, which assigns primitives to tiles by filling a memory structure named *Parameter Buffer*. The Parameter Buffer contains the vertex attributes of all the primitives of the frame being rendered. It also stores, for each tile, its *Display List*: a list of pointers to the attributes of the primitives that overlap that tile.

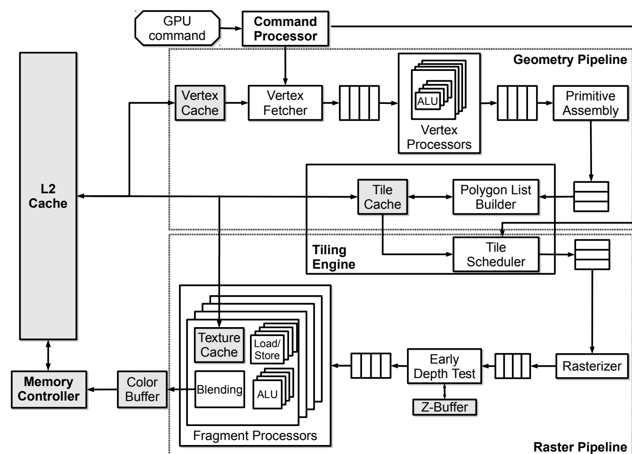


Figure 1. Assumed baseline architecture, similar to an ARM Mali-450.

The Raster Pipeline is triggered once all the geometry of the frame has been processed. Tiles are processed sequentially by fetching their primitives from the Parameter Buffer and dispatching them to the Rasterizer, which discretizes them into *fragments*, elements containing the information necessary to compute the color of a pixel. Fragments then can be queried for visibility in the Early Depth Test stage, which discards fragments that will not contribute to the color of the pixel because an already-processed fragment occludes them. The fragments passing the test proceed to the Fragment Processors, where their color is computed by user-defined *fragment shaders*. The output color may then be blended with previous computed values for that pixel (in case of transparencies) and the resulting value is stored in a local memory named *Color Buffer*, a rectangular array of pixel colors consisting of a red, green and blue component for each color. When all of the primitives of a tile have been rendered, the contents of the Color Buffer are flushed to main memory.

Rendering Elimination. Rendering Elimination [1] is a

technique that avoids the rendering of tiles that will produce the same color across consecutive frames. A signature of the primitives of each tile is computed and compared with the signature of the primitives the same tile had in the preceding frame. If the two signatures match, it is considered that the inputs for the Raster Pipeline have not changed between frames and, therefore, the output will also be the same. Consequently, the tile is not rendered and the colors of its pixels are reused with the ones computed in the previous frame. Figure 2 shows a block diagram of how the technique operates within the Graphics Pipeline. The Signature Buffer (1) is an on-chip lookup table that stores, for each tile, the signature of the previous frame and the in-progress signature of the current frame. Whenever a primitive reaches the end of the Geometry Pipeline, the signature of its vertex attributes is computed (2). In addition, for each tile that the primitive overlaps, the signature of the current frame for that tile is updated by combining the temporary value stored in the Signature Buffer entry with the computed signature of the primitive. When all the frame geometry has been processed, the Signature Buffer holds the final signatures for every tile in the current frame. Then, whenever a tile is scheduled to be rendered, its signatures for the current and the previous frame are read from the Signature Buffer and compared to decide whether or not the tile needs to be rendered (3).

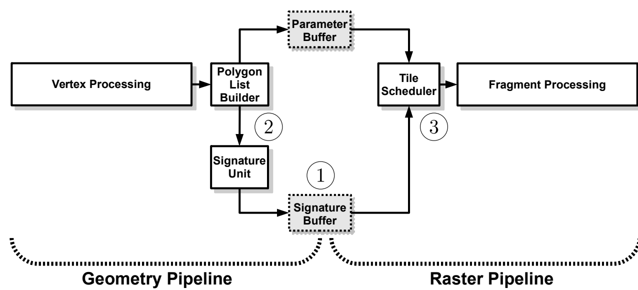


Figure 2. Rendering Elimination block diagram.

Early Depth Test. The Depth Test [4] is a per-fragment operation in which occluded fragments are discarded (do not write in the Color Buffer). To do so, the depth of the closest fragment to the camera is kept for every pixel in the screen in a memory structure named *Z Buffer*. After shading a fragment and before updating the Color Buffer, its depth value is checked against the stored value for that position in the Z Buffer: if the new fragment is closer to the camera than the previous one, the Color Buffer and Z Buffer entries are updated with the fragment's color and depth, respectively. Otherwise, the buffers are not updated. The *Early* Depth Test is an optimization supported by most modern GPUs in which fragments are discarded before shading them by applying the Depth Test prior to being dispatched to the Fragment Processors: a fragment will be shaded only if it passes this visibility test. However, not all primitives can benefit from this optimization because some shaders have the capability to change the visibility determined by the Early Depth Test. The most common operations to achieve such alterations are fragment discard (the execution of the shader prevents a fragment from progressing into the pipeline, thus causing an incorrect early update of the Z Buffer) and depth writing (a shader may generate a depth value which may cause a fragment considered visible to become occluded or vice versa).

Blending. Transparency is often computed by combining a translucent foreground color with a background color so as to create a filter of the color of the objects behind. To achieve this blending effect, an additional color component named *alpha* expressing the degree of opacity is included in each pixel. The alpha value ranges from 1 to 0, with 1 indicating that the fragment is fully opaque and 0 indicating full transparency. Whenever the fragment shader outputs a color for a fragment, it is combined with the existing value in the same entry of the Color Buffer in a way that depends on its alpha value. To properly render transparent objects, the value in the Color Buffer must correspond to the color of all the objects behind the transparent fragment, since the blending operation is generally not commutative. Therefore, the standard method for applying transparency is to first render the opaque geometry (thus taking advantage of the Depth Buffer capabilities) and then render all the translucent geometry in back-to-front order.

III. Early Visibility Resolution (EVR) of occluded primitives

Detecting occluded primitives early in the pipeline can prevent us to process them and avoid a significant amount of ineffectual work. However, this is a complex problem, so we rely on a simplification that estimates visibility with a low implementation cost. We exploit the fact that visibility tends to remain constant across consecutive frames: if a primitive is occluded in a frame, it will most likely be occluded in the following one.

A sufficient -but not necessary- condition for a primitive to be occluded in a tile is that the primitive is entirely located farther from the viewpoint than the farthest visible point in that tile. Based on that observation, we label primitives as occluded in a tile if they are farther than the farthest visible point (hereafter named *FVP*) for that tile in the previous frame. Whenever a frame finishes rendering, the visibility of the complete scene is known, so the depth of the FVP can be extracted for each tile.

Visibility is usually determined at a fragment level using the Early Depth Test. However, a large number of mobile 2D applications use the so-called Painter's Algorithm [5], where objects in the scene are drawn in back-to-front order. This way, a newly-processed opaque fragment always occludes previously-rendered fragments in the position it maps to

without the need of tracking depth information using the Z Buffer. Consequently, to determine the FVP for a tile, we must distinguish between primitives that write on the Z Buffer (*WOZ* primitives) and primitives that do not (*NWOZ*).

A. WOZ Primitives

All information regarding the visibility for these primitives is available in the Z Buffer when the tile is rendered. The per-tile FVP depth is computed as the maximum depth value stored in the Z Buffer (Z_{far}). A primitive is labeled as occluded in a given tile if its closest vertex (Z_{near}) to the viewpoint is farther than the FVP's depth from the previous frame.

The coarse granularity caused by comparing to a single Z_{far} value combined with the conservative Z_{near} comparison (which requires that all primitive points are beyond the FVP, not just those overlapping the tile) reduces the detection rate, since not all occluded primitives might be labeled as such. However, this way the primitives can be labeled as occluded for a tile earlier in the pipeline, with information available at the Polygon List Builder stage (vertex depths and tile binning of the primitive), without the need to either clip them to the boundaries of a tile or rasterize them. Note that Z_{far} is a single value per tile, so it is stored in an on-chip memory buffer at an acceptable energy and area overhead.

B. NWOZ Primitives

The visibility for these primitives is implicit in the rendering order and is, therefore, not resolved using the Z Buffer. However, by using a different mechanism, occluded primitives can still be detected in such scenes. During the sorting of primitives into tiles, we tally how many different draw commands have produced primitives that overlapped each particular tile and store that number in a *layer identifier* counter. The layer identifier of a tile starts at zero at the beginning of the frame and is increased by 1 whenever a primitive that belongs to a new command is sorted to that tile.

When a primitive is sorted to a tile, it is assigned the current layer identifier of that tile. Since opaque primitives in a layer partially or completely occlude layers laid under it, we can use those identifiers as depth information: primitives with higher layer identifiers are closer to the observer than primitives with smaller ones. Later on, after rasterization, layer identifiers are tracked for all the opaque visible fragments of a primitive in the *Layer Buffer*, which is a local structure akin to the Z Buffer.

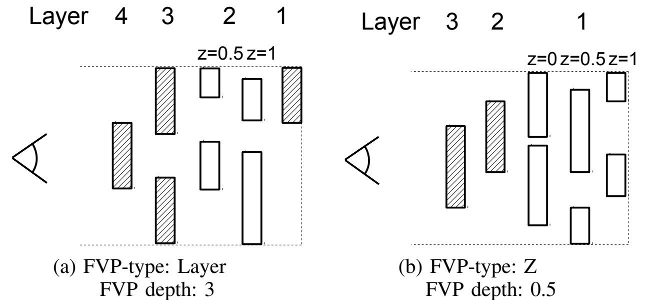
When a tile is completely rendered, all the information concerning its visibility is available in the Layer Buffer. The depth of the FVP corresponds to the minimum identifier stored in the Layer Buffer (L_{far}). A primitive is labeled as occluded in a tile if its assigned layer for the current tile is smaller than the tile's L_{far} from the previous frame.

C. Hybrid Scenes

A 3D scene is mainly composed of primitives that write in the Z Buffer, but it may also include primitives that do not. For instance, a batch of NWOZ primitives are sometimes drawn at the beginning of the scene as a background or at the end as a HUD[1]. Besides, it is common to find scenes with traditional alpha blending, where geometry is rendered in two steps. In the first one, the opaque geometry is rendered. In the second step, the translucent primitives are processed in back-to-front order. Translucent primitives are NWOZ because by definition they are not occluders, so they must not update the Z Buffer.

WOZ primitives are also assigned a layer identifier to help compare its age relative to NWOZ primitives. However, since resolving visibility among WOZ primitives themselves is not determined by their relative age but by comparing their explicit depth values, we can assign the same layer identifier to all of the WOZ primitives in a batch. If two primitives, one being a WOZ and the other being an opaque NWOZ, overlap the same pixel, visibility is resolved by comparing their relative age, i.e., by determining which one was rendered last.

The FVP depth of a tile may be either Z_{far} or L_{far}, depending on whether the FVP belongs to a WOZ or a NWOZ primitive, respectively. After computing L_{far}, we determine to which type of primitive it belongs and store the proper FVP depth value (either Z_{far} or L_{far}) for the tile. A boolean value termed *FVP-type* is then set to indicate whether the stored FVP corresponds to a WOZ or a NWOZ primitive.



(a) FVP-type: Layer
FVP depth: 3

(b) FVP-type: Z
FVP depth: 0.5

Figure 3. FVP depth computation in tiles with both WOZ primitives (white) and NWOZ primitives (striped).

Figure 3 illustrates how the FVP of a tile is computed in the presence of both WOZ and NWOZ primitives. A tile is viewed in a top-down perspective with the location of its primitives represented as rectangles. The observer of the scene placed on the left, i.e., the right corner is farther. The top of the figure displays the layer identifiers for all primitives as well as the Z value for WOZ primitives.

[1]HUD is short for Head-Up Display, a visual overlay used to present information to the user.

In the scenario presented in Figure 3a, Layer 1 is completely occluded by Layer 2, whereas Layer 2 is completely occluded by Layers 3 and 4. Layer 3 is visible, so the L_{far} of the tile is 3. Since Layer 3 belongs to NWOZ primitives, the FVP depth of the tile is its L_{far} and a corresponding FVP-type that indicates that the FVP is a layer is stored.

In the scenario presented in Figure 3b, Layer 1 is visible, so the L_{far} of the tile is 1. Since Layer 1 belongs to WOZ primitives, the FVP depth corresponds to the tile's Z_{far}. Primitives with a depth value of 1 are occluded by primitives with smaller depth values, while primitives with a depth value of 0 do not completely occlude primitives with a depth value of 0.5. Thus, the Z_{far}, and consequently the FVP depth, of the tile is 0.5. The FVP-type of the tile is set to indicate that the FVP is a Z value. A primitive is labeled as occluded if one of the following two scenarios occurs:

- The FVP in the previous frame is NWOZ and the layer assigned to the primitive is lower than L_{far}
- The primitive and the FVP in the previous frame are WOZ, and the primitive's Z_{near} is farther than Z_{far}.

IV. Removing Innefectual Computations with EVR

In this section, we present two optimizations that leverage the presented EVR mechanism for early detection of occluded primitives to avoid ineffectual computations and memory accesses in the Graphics Pipeline.

A. Overshading Reduction

Overshading occurs when a pixel is shaded multiple times because several primitives overlap it. If an opaque primitive writes into an already-shaded pixel, the resources devoted to the previous color computation have been wasted because it has no effect in the final image. Note that some overshading cannot be avoided, such as the one produced by translucent primitives. As introduced before, GPUs try to reduce overshading by employing an Early Depth Test which avoids shading a fragment if a closer, opaque fragment has already been processed. Although this mechanism can eliminate a significant fraction of overshading, it is heavily dependent on the order that fragments are processed because it can only discard fragments which are hidden by those already processed. A direct solution to the overshading problem would be for the application to sort the opaque primitives in a front-to-back order. However, many of these software-based approaches require building costly spatial hierarchical data structures to render the scene from any single viewpoint. They are only effective on "walkthrough" applications where the entire scene is static and only the viewer moves through it, because the overheads can be amortized over a large number of frames. Furthermore, such application-level sorting is often challenging due to cyclic overlaps among objects or objects containing geometry that occludes parts of the same object.

Some applications perform a preliminary depth pre-pass (either in hardware or in software), where some or all of the geometry is rendered using a simple *shader* that only writes into the Z Buffer. After that, the actual render pass is executed but now having perfect or near-perfect visibility information in the Z Buffer, which greatly increases the number of fragments discarded by the depth test. Despite the improved efficacy of the depth test, the overhead of the additional render pass is very high and often offsets its potential benefits.

In this paper we propose to use the speculative visibility determination mechanism described in Section III to *dynamically reorder* opaque primitives so as to render primitives that are likely to be occluded after primitives that are likely to be visible without the need of an additional render pass. The reordering is performed in the Polygon List Builder stage, when primitives are sorted into tiles. In the baseline configuration, for each primitive the Parameter Buffer is updated as follows: the primitive's attributes are stored in memory and a pointer to those attributes is written into the Display List of each tile. Then, whenever a tile is rendered, its Display List is accessed and the pointers to primitives are dereferenced to access their attributes to rasterize them.

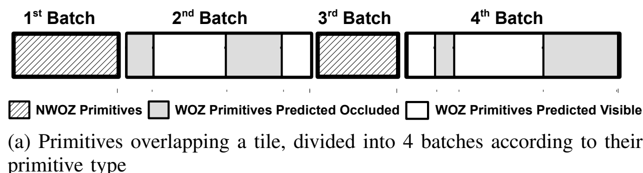
The proposed reordering mechanism divides the Display List of every tile into two lists. Tiles are rendered by fetching initially all the primitives from the first list and then the primitives from the second list. Whenever a primitive is sorted into a tile, its attributes are stored into the Parameter Buffer the same way as in the baseline. The pointer to those attributes, on the other hand, is stored on one of the lists depending on the type of primitive and its predicted visibility, according to Algorithm 1. This algorithm only

Algorithm 1 Reordering Algorithm based on FVP

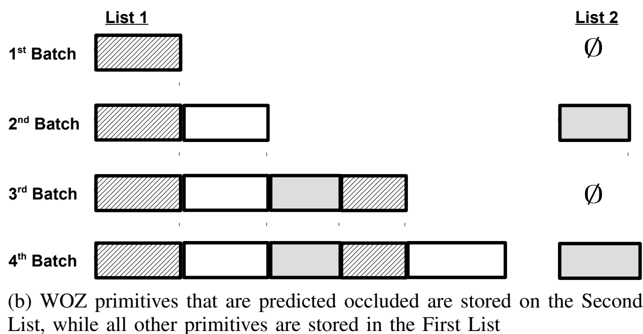
```
if Primitive is WOZ then
    if Predicted visible then
        Append into First List
    else
        Append into Second List
    end if
else                                        ▷ NWOZ Primitive
    if Second List not empty then
        Move Second List to the end of the First List
    end if
    Append into First List
end if
```

reorders opaque WOZ primitives among themselves, while preserving the order of NWOZ primitives against themselves and against WOZ primitives. Reordering WOZ primitives does not incur in rendering errors: NWOZ primitives are not reordered and all WOZ primitives perform the depth test as usual, which maintains the correctness of the result produced by such primitives regardless of the order in which

they are rasterized.



(a) Primitives overlapping a tile, divided into 4 batches according to their primitive type



(b) WOZ primitives that are predicted occluded are stored on the Second List, while all other primitives are stored in the First List

Figure 4. Reordering algorithm example.

To illustrate Algorithm 1, let us consider Figure 4a, showing 4 different batches of primitives of a particular tile. The two WOZ batches include primitives that are predicted to be visible and primitives that are predicted to be occluded. Figure 4b shows how they are reordered.

The first processed batch is an NWOZ batch. No actions are performed with the second list since it is empty, and all the primitives in the batch are appended to the first list. Next, there is a WOZ batch, whose primitives are appended to the first list if they are predicted to be visible in the tile and appended to the second list otherwise. When the following NWOZ batch arrives, all the primitives in the second list are moved to the end of the first list and then the primitives of the batch are appended to the first list. Finally, another WOZ batch is processed, whose primitives are again appended to the two lists according to their predicted visibility.

This reordering technique is highly effective at reducing overshading because if the visibility prediction is correct, the early depth test is able to discard more fragments, which reduces the amount of computation and memory accesses devoted to occluded fragments. Note that this scheme does not introduce any error as commented above. Visibility mispredictions simply imply a loss of culling effectiveness in the Early Depth Test.

B. Rendering Elimination Improvement

Rendering Elimination [1] is a technique that detects tiles that produce the same color across adjacent frames. To do so, when primitives are sorted into tiles at the end of the Geometry Pipeline, a signature per tile is incrementally computed on-the-fly with the attributes of all primitives overlapping each tile. Then, when the Raster Pipeline starts processing a tile, its signature computed for the current frame is compared against the signature computed in the previous frame: if both signatures match, the tile is not rendered since it will produce the same colors as in the previous frame. Rendering Elimination requires all attributes from all primitives of a tile to be exactly the same as in the previous frame to detect redundancy. However, in the case that only occluded primitives change their attributes, the tile's colors will be the same as for the preceding frame, making Rendering Elimination not able to detect and eliminate such frame-to-frame redundancy. In this paper, we also propose using the approximate visibility resolution mechanism described in Section III to compute signatures only with visible primitives so as to improve the effectiveness of Rendering Elimination's tile redundancy detection.

Table I
VISIBILITY CASUISTRY

Scenario	Frame i	Frame i+1
A	Visible	Visible
B	Visible	Occluded
C	Occluded	Occluded
D	Occluded	Visible

In the baseline operation of Rendering Elimination, a lookup table named *Signature Buffer* stores one CRC32 per tile. Whenever a primitive is sorted, the CRC32 of the attributes of its vertices is computed. Then, for all the tiles that the primitive overlaps, the corresponding Signature Buffer entry is read and updated by combining the CRC32 value of the entry with the CRC32 value of the sorted primitive.

Using the visibility prediction scheme proposed in Section III, we propose to extend the Rendering Elimination technique as follows. For each sorted primitive, its depth is compared against the depth of the FVP in the previous frame for each tile it overlaps. If the primitive is predicted to be occluded in a tile, the Signature Buffer entry for that tile is not updated with the CRC32 of the primitive. As it will be shown in the Results section, utilizing the FVP depth allows for a significant increase in redundant tile detection. Moreover, the proposed optimization does not produce any rendering errors. Table I presents the four possibilities regarding the resolved visibility of a primitive (either visible or occluded) across two consecutive frames.

For scenarios A and B the optimization behaves like the baseline Rendering Elimination: since the primitive was visible in the tile in Frame i, it is considered for the signature of the tile in Frame $i+1$, regardless of its final visibility. Note that, in scenario B, the primitive will be occluded and, therefore, will not be considered in the signature in Frame $i+2$. This is the case for scenarios C and D.

Scenario C is the case that improves over the baseline: since the occluded primitive does not affect the final colors of the tile, not considering the primitive for the signature

enhances redundancy detection while not generating errors.

Finally, scenario D does not cause rendering errors because for a primitive P (occluded in Frame i) to be visible in Frame $i+1$, at least one of the following two conditions must hold:

i) P has moved closer to the camera than the farthest depth of the tile in the previous frame. In that case, P will be added to the signature of the tile. Since it was not included in the signature of the previous frame, the signatures will differ and the tile will be rendered.

ii) All the primitives that occluded P have moved (or are not rendered) so that in Frame $i+1$ they do not totally occlude P. In that case, the attributes of the occluder primitives must have changed and the signature will be different: even if P is not considered for the signature, the tile will be rendered.

V. Implementation

In this section, we describe the extra hardware required to implement the proposed early visibility resolution mechanism, which basically consists of additional units to compute and store the FVP (farthest visible point) for all the tiles in the frame. Figure 5 shows how they are integrated into a TBR GPU.

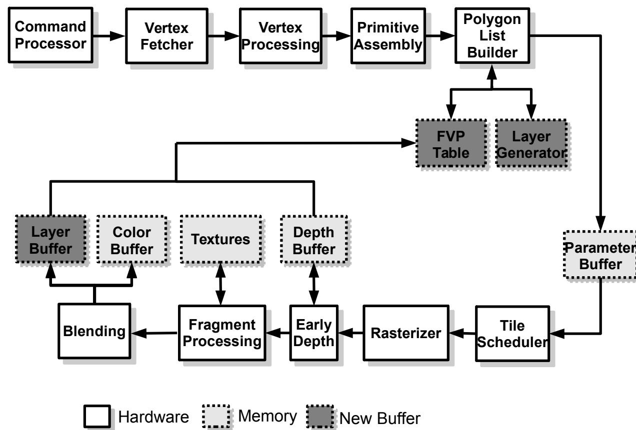


Figure 5. Graphics Pipeline including the structures needed to implement FVP computation.

A. Layer Generator Table

As discussed in Section III, layers are tracked to emulate depth among NWOZ primitives. Assigning a layer identifier to a primitive requires to address the following issues.

First, since the layer identifier is intended to count the number of objects (draw commands) whose primitives have overlapped a given tile so far, each tile must have its independent layer counter. Of course, for a given tile, all the primitives of the same command are assigned the same layer identifier, although that layer may differ from one tile to another.

Second, since WOZ primitives update their depth into the Z Buffer, layer identifiers do not provide any information among primitives in a WOZ batch: the same identifier can be assigned to all WOZ primitives in a batch for a given tile.

We propose to employ a small, on-chip LUT which we call *Layer Generator Table* in order to manage the association of layer identifiers to primitives. This table has one entry per tile, and each entry contains three fields:

1) Last identifier of a command that produced a primitive that overlapped the tile.
2) Last layer assigned to a primitive that overlapped the tile.
3) Last type of primitive (WOZ/NWOZ) that overlapped the tile.

Using the Layer Generator Table (LGT) we can assign a layer to every primitive in all the tiles it overlaps during the Polygon List Builder stage. Whenever a primitive is sorted into a tile, the LGT entry for that tile is checked. If the stored command identifier is the same as the primitive's command identifier, it means that the primitive belongs to the same layer as the last primitive sorted into that tile. Consequently, the primitive is assigned the layer stored in the entry. After sorting a primitive into a tile, it updates the *last type of primitive* field in the LGT (a binary value: NWOZ or WOZ).

On the other hand, if the stored command identifier is different to the primitive's command, the layer may be increased depending on the type of primitive. NWOZ primitives always increase the layer number whereas for WOZ primitives the layer is only increased if the previous primitive was NWOZ. Finally, the LUT entry is updated with the new command identifier and the new layer value if they have changed.

The layer identifier of a primitive is stored in the Parameter Buffer, as any other attribute. This way, layers can be assigned to all the fragments of the primitive at rasterization time.

B. Layer Buffer

The farthest visible layer of a tile can be obtained by computing the minimum visible layer of all its pixels. Since tiles are relatively small (e.g. 16x16 pixels) we can use an on-chip buffer to keep track of per-pixel information for an entire tile. This buffer, which we call the *Layer Buffer*, has one entry for every pixel of the tile being rendered (just as the Z Buffer or the Color Buffer) that stores the visible layer for that pixel.

The Layer Buffer is updated during the Blending stage, when the final fragment opacity is already determined. To detect opacity, we use the same alpha value that fragments employ to blend with colors previously written in the Color Buffer. If the alpha factor is exactly 1, the fragment is opaque and its layer is written into the Layer Buffer. Otherwise,

since the fragment is translucent and does not completely occlude layers behind it, the Layer Buffer is not updated.

During the blending stage, each fragment of a WOZ primitive stores its layer identifier in a register named *ZR*, so that it identifies the layer of the last visible WOZ primitive and may be used to distinguish, at the end of rendering a tile, if its FVP corresponds to a WOZ or a NWOZ primitive, i.e., the FVP-type of the tile. The value of ZR is compared to L_{far} when the tile finishes rendering: if the two values are equal, the FVP-type of the tile is WOZ. Otherwise, it is NWOZ.

C. FVP Table

In order to predict if a primitive is likely to be visible in a tile we use the tile's FVP depth of the previous frame, so the entire set of per-tile FVP depths must be stored. Such information is maintained in a structure that we call *FVP Table*. The FVP Table has one entry per tile, with each entry containing the previous frame's FVP depth for that tile. Each entry in the table also stores the *FVP-type* to indicate whether the type of data stored is a Z or a layer identifier.

Whenever a tile finishes rendering, its FVP-type is determined. If the FVP depth belongs to a NWOZ primitive, the FVP Table entry for the tile is updated with L_{far}, setting its FVP-type bit. Otherwise, the FPV entry for the tile is updated with Z_{far} and the entry's FVP-type bit is cleared.

VI. Evaluation Methodology

A. Simulator infrastructure

We employ the Teapot simulation framework [6] to evaluate our proposal. Teapot runs unmodified Android applications for mobile platforms and obtains GPU performance and energy consumption statistics. Table II lists the parameters used in the simulations in order to model an architecture resembling the ARM Mali-450 GPU [2], the most widely used GPU architecture nowadays, with almost 20% of the mobile GPU market [3].

The traces that feed the Teapot cycle-accurate simulator are obtained via a two-step process: First, an Android application is run in the Android emulator [7], and every OpenGL command sent to the GPU is intercepted and stored in a file. Second, those OpenGL commands are fed to the software renderer included in Gallium3D [8], which is instrumented to generate a trace containing all the information needed to guide the cycle accurate execution, such as vertex data, shader instructions or memory addresses of the different stages of the Graphics Pipeline. For each application, we simulate 60 consecutive frames from the original application run.

McPAT [9] is used to estimate the energy consumption of the GPU. All the additional hardware required for the proposed technique (Layer Generator Table, FVT table, Layer Buffer, comparators and registers) is modelled using McPAT's components (SRAMs, Registers, MUXes, XORs).

Table II
GPU Simulation Parameters.

Baseline GPU Parameters	
Tech Specs	400 MHz, 1 V, 32 nm
Screen Resolution	1196x768
Tile Size	16x16 pixels
Main Memory	
Latency	50-100 cycles
Bandwidth	4 B/cycle (dual channel LPDDR3)
Size	1 GB
Queues	
Vertex (2x)	16 entries, 136 bytes/entry
Triangle, Tile	16 entries, 388 bytes/entry
Fragment	64 entries, 233 bytes/entry
Caches	
Vertex Cache	64 bytes/line, 2-way associative, 4 KB, 1 bank, 1 cycle
Texture Caches (4x)	64 bytes/line, 2-way associative, 8 KB, 1 bank, 1 cycle
Tile Cache	64 bytes/line, 8-way associative, 128 KB, 8 banks, 1 cycle
L2 Cache	64 bytes/line, 8-way associative, 256 KB, 8 banks, 2 cycles
Color Buffer	64 bytes/line, 1-way associative, 1 KB, 1 bank, 1 cycle
Depth Buffer	64 bytes/line, 1-way associative, 1 KB, 1 bank, 1 cycle
Non-programmable stages	
Primitive assembly	1 triangle/cycle
Rasterizer	16 attributes/cycle
Early Z test	32 in-flight quad-fragments, 1 Depth Buffer
Programmable stages	
Vertex Processor	1 vertex processor
Fragment Processor	4 fragment processors
Additional hardware	
Layer Generator Table	3600 entries, 3 bytes/entry
FVP Table	3600 entries, 4 bytes/entry
Layer Buffer	64 bytes/line, 1-way associative, 1 KB, 1 bank, 1 cycle

Regarding system memory, Teapot employs DRAMSim2 [10].

B. Benchmarks

Table III presents the benchmarks employed to test the proposed approach, which correspond to twenty unmodified Android graphics applications. All of these applications have millions of downloads according to Google Play [11], with some of them surpassing 500 million downloads. The set of benchmarks includes games of the most popular genres in the mobile segment [12], such as puzzle, arcade and simulation. The benchmarks also contain a variety of workloads: completely 2D scenes such as *cde* or *wmw*, scenes with simple 3D models such as *ata* or *tib*), and

scenes with more sophisticated models such as *300* or *mst*. Applications classified as *3D* contain both WOZ and NWOZ primitives, while applications classified as *2D* only contain NWOZ primitives.

Table III
BENCHMARK SUITE.

Benchmark	Alias	Genre	Type
300: Seize your glory	300	Action	3D
Air Attack	ata	Arcade	3D
Crazy Snowboard	csn	Arcade	3D
Modern Strike	mst	First Person Shooter	3D
Temple Run	ter	Platform	3D
Tigerball	tib	Physics Puzzle	3D
Angry Birds	abi	Puzzle	2D
Armymen	arm	Strategy	2D
Avenger Legends	ale	Strategy	2D
Candy Crush Saga	ccs	Puzzle	2D
Castle Defense	cde	Tower Defense	2D
Clash of Clans	coc	MMO Strategy	2D
Cut the Rope	ctr	Puzzle	2D
Dude Perfect	dpe	Puzzle	2D
Hayday	hay	Simulation	2D
Hopeless	hop	Action Survival	2D
Magic Touch	mto	Arcade	2D
Redsun	red	Strategy	2D
Where's my water	wmw	Puzzle	2D
World of goo	wog	Physics Puzzle	2D

VII. EXPERIMENTAL RESULTS

Figure 6 shows the energy consumption of the GPU-Memory system normalized to the Baseline GPU for our set of benchmarks. The proposed optimizations that leverage an early prediction of the visibility achieve a 43% reduction of energy consumption on average. Energy savings are obtained in all benchmarks, with maximums of more than 80% (*cde*, *dpe*). Figure 7 shows the execution time, divided into Geometry and Raster pipelines, normalized to the baseline. On average, the proposed techniques achieve 39% execution time reduction, with maximums of more than 70% (*ccs*, *cde*, *dpe*).

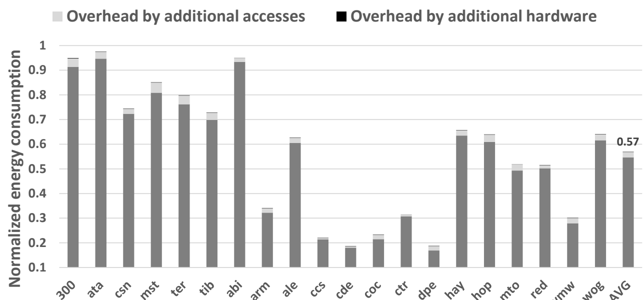


Figure 6. Energy consumption of our EVR proposal normalized to the Baseline GPU.

The energy overheads are mainly due to additional writes to the Parameter Buffer to store the layer identifiers. This overhead is quite moderate, 2.1% on average, as it can be seen in Figure 6. Figure 6 also illustrates that the additional hardware added by the proposed mechanism incurs in only 1.2% energy consumption overhead on average: the structures needed to manage the FVP information (Layer Generator Table, Layer Buffer and FVP Table) generate 0.5% additional static and dynamic energy consumption while the LUTs needed to implement Rendering Elimination contribute to an additional 0.7% energy consumption. The computation of Rendering Elimination's tile signatures incurs in time overhead in the Geometry Pipeline whenever a primitive overlaps a large number of tiles, since the pipeline is stalled waiting for all signatures to be sequentially updated. Figure 7 reports such overheads in execution time which, on average, represents 0.5% of the total.

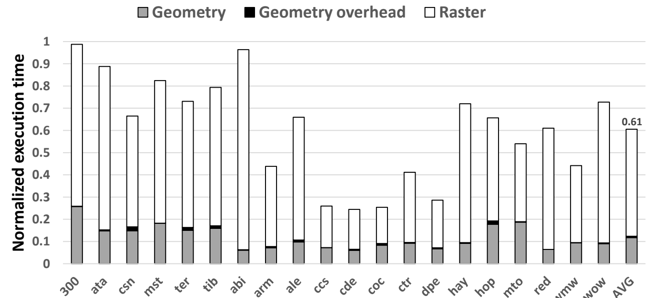


Figure 7. Execution time of our EVR proposal normalized to the Baseline GPU.

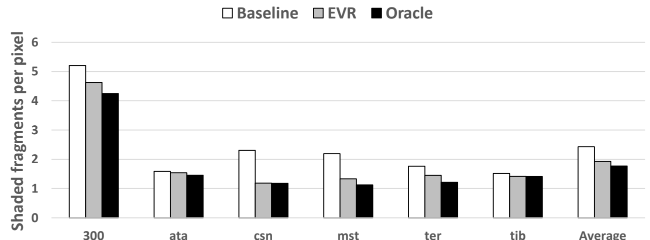


Figure 8. Comparison of the number of shaded fragments per pixel among the baseline GPU, our EVR proposal and an oracle.

These important reductions in energy consumption and execution time are mainly produced by avoiding the processing of ineffectual fragments (fragments that are occluded or are the same as in the previous frame). Figure 8 shows the number of fragments shaded per pixel using the proposed early visibility resolution mechanism to reorder primitives (*EVR*) compared to the baseline for our set of 3D benchmarks. We also compare our scheme with an oracle approach (which ideally assumes that the Z Buffer is initialized with the final visibility of the tile –the final depth values– before it is executed). EVR significantly reduces (20%) the number of vainly shaded fragments, and its results are close to those obtained by an oracle approach. EVR cannot reach the oracle because of its approximate nature. First, it uses visibility information of the previous frame, which may have changed.

Second, it estimates visibility at a primitive-level, while the oracle resolves visibility at the finest possible granularity: fragment level.

Moreover, a large fraction of tiles are detected to be redundant and its execution is completely bypassed, avoiding not only the Fragment Shader stage for all their fragments, but also the rest of the stages of the Raster Pipeline for those tiles. Figure 9 shows the percentage of detected redundant tiles –producing the same colors as the previous frame– for our set of benchmarks. Results are shown for the baseline Rendering Elimination (RE), the proposed EVR-aided Rendering Elimination (*EVR*), and an oracle setup that can perfectly identify all tiles of a frame that are equal to those of the previous frame.

Figure 9. Percentage of detected equal tiles by EVR, compared to RE and an oracle.

On average, EVR avoids the rendering of 54% of the tiles, reducing 5% more tiles than the baseline RE. Predicting occluded primitives and not adding them to the tile's signature allows the detection of equal tiles in benchmarks in which RE was hardly effective, such as *300* or *mst*. The majority of these tiles that are identified as redundant corresponds to portions of the screen in which WOZ primitives are covered by NWOZ primitives in the form of a HUD. In most cases the scene contains objects that are in motion but completely occluded by NWOZ geometry. The layer-based visibility scheme allows EVR to identify additional redundant tiles in benchmarks with a high degree of redundancy detected by RE, such as *cde* or *mto*. In some applications such as *hay* or *wmw*, the additional tiles eliminated exceeds 10%, with a maximum of 30%.

The additional redundant tiles detected results in an average 10% reduction of energy consumption compared to the baseline RE, as presented in Figure 10. The early visibility resolution incurs in some overheads, which are grouped together in the figure. In the Geometry Pipeline, the Layer Generator and FVP tables are accessed to generate a layer identifier, which is stored in the Parameter Buffer. In the Raster Pipeline, the layer identifiers are read and written into the Layer Buffer. When a tile finishes its rendering, its FVP is computed by accessing the Z and Layer Buffers, and the result is stored in the FVP Table. Although the required

Figure 10. Energy consumption of our EVR proposal normalized to RE.

sequential check of the two tables for each primitive and each tile to which it is mapped could incur in some time overhead, it is more than offset by the reduction in number of primitives that can skip the signature computation step since they are occluded. This is shown in Figure 11, which compares execution time, broken down into Geometry and Raster Pipelines, for the Baseline GPU, RE and our proposed EVR approach for early visibility resolution. Note that RE has to read the temporary hashes held in the Signature Buffer for all the tiles that a primitive overlaps, shift them as many bytes as the size of the primitive and, finally, combine them with the hash of the primitive to produce a new signature for the tile. This process is avoided in our scheme if a primitive is determined to be occluded in a tile, since EVR marks it not to be included into the tile's signature. The only exception where our EVR scheme does not reduce Geometry cycles is *hop*, a benchmark that has very few primitives concentrated in a reduced amount of tiles. Consequently, the average number of primitives to combine into the signature of a tile is small, limiting the benefits of occluded primitive detection. On average, our EVR mechanism reduces the execution time in the Geometry Pipeline by 4% with respect to RE.

Moreover, Rendering Elimination may induce time and energy overheads in benchmarks where not enough redundant tiles are detected to offset the signature computation, such as *300* or *mst*. On the other hand, our scheme is more effective at detecting redundant tiles and, for non-redundant tiles specially in 3D benchmarks, the primitive reordering increases the efficiency of the Early Depth Test, which reduces the amount of computations in the fragment processors. This results in overall speedups for all benchmarks.

VIII. Related Work

Occlusion culling is a fundamental problem in computer graphics, with an extensive literature spanning more than three decades. In this section, we focus on the most influential works and the ones most related to our proposal.

The Z Buffer [4] is the *de facto* standard to resolve visibility due to its effectiveness and low implementation cost. The Hierarchical Z Buffer [13] is a variation of the baseline technique in which a depth pyramid is used to test visibility. The base level of the pyramid is the Z Buffer

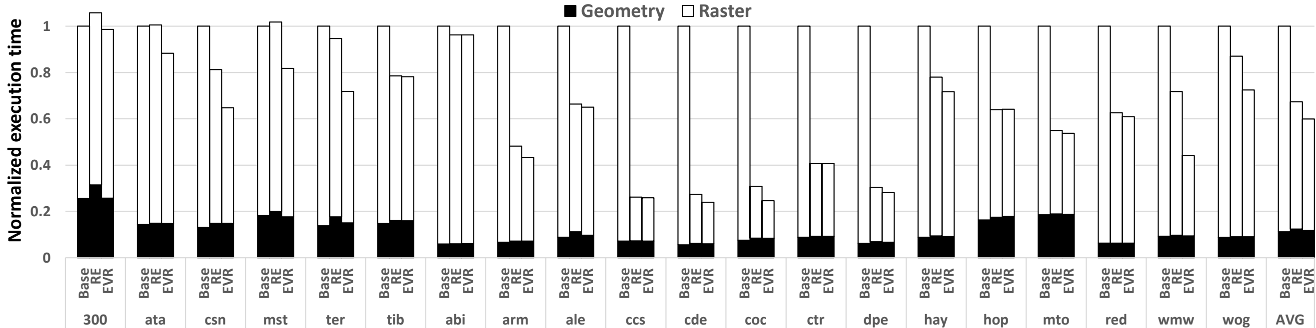


Figure 11. Execution time comparison of our EVR proposal and RE against the baseline GPU.

and higher levels are constructed by combining the depth of four pixels at the next lower level, typically by choosing the farthest one. Entire primitives can be discarded without accessing the Z Buffer by comparing their nearest depth against the values in higher levels in the pyramid. Our EVR proposal also compares the depth of primitives to a FVP depth, which would correspond to the top of the pyramid of the Hierarchical Z Buffer. However, the FVP depth contains final visibility information which allows, unlike the Z pyramid, to detect primitives that will be occluded by others processed later. Moreover, the FVP depth includes more information than just the top of the Z pyramid, allowing the detection of occluded NWOZ primitives that do not use the Z buffer.

Z-Prepass [14] draws the scene in two steps: first the geometry is rendered using simple fragment shaders only to quickly fill the Z Buffer. Later, the geometry is rendered again with proper shaders, but now with perfect visibility information in-place, so that all occluded fragments can be discarded. This additional render pass incurs in significant overheads, which may not always be offset by the increase in the fragments discarded in the Early Depth Test. By working at a coarser granularity (primitive instead of fragment), EVR does not need to perform the pre-pass but still achieves results comparable to having complete visibility information.

The concept of *layers* has been previously adopted in the context of occlusion culling, most notably in Depth Peeling [15], an algorithm that renders geometry multiple times, peeling off the surface layer at each pixel in each pass. Recently, Andersson et al. [16] leveraged a two-layer representation of depths to avoid bandwidth spent in updating the Hierarchical Z Buffer, while in the work of Scheckel and Kolb [17] layers are used in combination with the alpha parameter to completely cull transparent fragments. Unlike EVR, these approaches cannot combine visibility information of both WOZ and NWOZ primitives for a better visibility determination.

Computing visibility at a fragment level (known as image-precision [18]) is useful to solve certain problems, such as circular dependencies. However, resolving visibility at a coarser grain could reduce the number of computations needed. Hardware occlusion queries [19] are a feature that allow the user to query simple geometry (such as the bounding volumes) against the current contents of the Z Buffer. The query counts the number of fragments that pass the Z Test and the result allows the application to avoid rendering entire objects at the expense of CPU-GPU synchronization. Furthermore, as with the Early Depth Test, in order for occlusion queries to perform well, both objects and queries must be sent in front-to-back order. EVR is transparent to the application in both axis: it does not require neither synchronization nor ordering.

Multiple works reduce overshading by means of reordering the primitives that make up a scene. Govindaraju et al. [20] propose an algorithm to sort non-overlapping objects in either front-to-back or back-to front order from a given viewpoint. In the work of Chen et al. [21], static objects are preprocessed in order to create a depth-sorted list of primitives for every possible viewpoint. The approach of Han and Sander [22] also preprocesses objects to create several sorted lists that are indexed at runtime to ensure optimal order, but they consider movement by taking into account not only viewpoints but also several key frames. Weber and Stamminger [23] use a graph representation of dependencies in animated scenes with a fixed camera to sort primitives accordingly, and leverage frame-to-frame coherence to merge different graphs and keep the overall structure manageable. VRO [24] also takes advantage of temporal coherence to reorder objects entirely in hardware and reduce overshading. Unlike these approaches, EVR is applied at a finer granularity (primitive instead of object), does not need to perform any preprocess and can also be employed in interactive scenes, not just animated ones.

IX. Conclusions

We have presented a mechanism to determine visibility in early stages of the Graphics Pipeline based on exploiting frame coherence. Since consecutive frames tend to be very similar, we use the information of a frame to estimate the visibility for the following one.

The proposed technique collects the depth of the farthest visible primitive of every tile whenever its rendering process is complete. For each overlapped tile, the depth of a primitive

is compared against the depth stored for that tile in the preceding frame. If it is farther, the primitive is occluded.

This paper demonstrates the benefits of early visibility prediction to remove ineffectual computations, by increasing the effectiveness of the Early Z Test, a commonly used technique in contemporary GPUs, and Rendering Elimination, a recently proposed technique to exploit redundant computations. The former works at pixel granularity and the latter works at tile granularity.

Using the predicted visibility information, opaque primitives whose visibility is resolved using the Z Buffer are reordered such that primitives predicted as visible are rendered first, which avoids the shading of occluded fragments. Besides, primitives predicted to be occluded are not considered when generating the signature used by Rendering Elimination to identify tiles that are equal to the ones in the previous frame. That increases the number of tiles that are identified as redundant and, consequently, whose rendering can be avoided.

Our technique provides average speedups of 39% and energy savings of 43% for a set of commercial Android applications. The reorder mechanism achieves overshading reductions comparable to having a Z Buffer filled with perfect visibility information without requiring any additional render pass to compute depths. On the other hand, by improving the tile redundancy detection of Rendering Elimination, the raster pipeline of the GPU skips the rendering of more than half of the tiles on average.

X. Acknowledgements

This work has been partially supported by the Spanish State Research Agency under grant TIN2016-75344-R (AEI/FEDER, EU) and by the Generalitat de Catalunya under the AGAUR-FI program.

References

[1] M. Anglada, E. de Lucas, J.-M. Parcerisa, J. L. Aragón, P. Marcuello, and A. González, "Rendering elimination: Early discard of redundant tiles in the graphics pipeline," in *Proceedings of the 25th IEEE International Symposium on High-Performance Computer Architecture*, IEEE, 2019.

[2] "Arm mali-450 gpu." https://developer.arm.com/products/graphics-and-multimedia/mali-gpus/mali-450-gpu, accessed December 10, 2018.

[3] "Hardware gpu market." http://hwstats.unity3d.com/mobile/gpu.html, accessed December 10, 2018.

[4] E. Catmull, "A subdivision algorithm for computer display of curved surfaces," tech. rep., Utah University Salt Lake City School of Computing, 1974.

[5] "Painter's algorithm." https://www.siggraph.org/education/materials/HyperGraph/scanline/visibility/painter.htm, accessed December 10, 2018.

[6] J.-M. Arnau, J.-M. Parcerisa, and P. 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*, pp. 37–46, ACM, 2013.

[7] "Android studio." https://developer.android.com/studio/index.html, accessed December 10, 2018.

[8] "Gallium3d." https://www.freedesktop.org/wiki/Software/gallium, accessed December 10, 2018.

[9] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. 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*, pp. 469–480, 2009.

[10] P. Rosenfeld, E. Cooper-Balis, and B. Jacob, "Dramsim2: A cycle accurate memory system simulator," *IEEE Computer Architecture Letters*, vol. 10, no. 1, pp. 16–19, 2011.

[11] "Google play." https://play.google.com, accessed December 10, 2018.

[12] "Mobile game statistics." https://medium.com/@sm_app_intel/new-mobile-game-statistics-every-game-publisher-should-know-in-2016-f1f8eef64f66, accessed December 10, 2018.

[13] N. Greene, M. Kass, and G. Miller, "Hierarchical z-buffer visibility," in *Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques*, pp. 231–238, ACM, 1993.

[14] "Early z rejection." https://software.intel.com/en-us/articles/early-z-rejection-sample, accessed December 10, 2018.

[15] C. Everitt, "Interactive order-independent transparency," *White paper, NVIDIA*, vol. 2, no. 6, p. 7, 2001.

[16] M. Andersson, J. Hasselgren, and T. Akenine-Möller, "Masked depth culling for graphics hardware," *ACM Transactions on Graphics*, vol. 34, no. 6, p. 188, 2015.

[17] S. Scheckel and A. Kolb, "Min-max mipmaps for efficient 2d occlusion culling," in *Conference on Computer Graphics, Visualization and Computer Vision*, 2016.

[18] I. E. Sutherland, R. F. Sproull, and R. A. Schumacker, "A characterization of ten hidden-surface algorithms," *ACM Computing Surveys*, vol. 6, no. 1, pp. 1–55, 1974.

[19] O. Mattausch, J. Bittner, and M. Wimmer, "Chc++: Coherent hierarchical culling revisited," in *Computer Graphics Forum*, vol. 27, pp. 221–230, Wiley Online Library, 2008.

[20] N. K. Govindaraju, M. Henson, M. C. Lin, and D. Manocha, "Interactive visibility ordering and transparency computations among geometric primitives in complex environments," in *Proceedings of the 2005 Symposium on Interactive 3D Graphics and Games*, pp. 49–56, ACM, 2005.

[21] G. Chen, P. V. Sander, D. Nehab, L. Yang, and L. Hu, "Depth-presorted triangle lists," *ACM Transactions on Graphics*, vol. 31, no. 6, p. 160, 2012.

[22] S. Han and P. V. Sander, "Triangle reordering for reduced overdraw in animated scenes," in *Proceedings of the 20th ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, pp. 23–27, ACM, 2016.

[23] C. Weber and M. Stamminger, "Topological triangle sorting for predefined camera paths," in *Proceedings of the Conference on Vision, Modeling and Visualization*, pp. 153–160, Eurographics Association, 2016.

[24] E. De Lucas, P. Marcuello, J.-M. Parcerisa, and A. Gonzalez, "Visibility rendering order: Improving energy efficiency on mobile gpus through frame coherence," *IEEE Transactions on Parallel and Distributed Systems*, 2018.