Exploiting Frame Coherence in Real-Time Rendering for Energy-Efficient GPUs

UNIVERSITAT POLITÈCNICA
DE CATALUNYA
BARCELONATECH

Martí Anglada Sánchez

Doctor of Philosophy

Department of Computer Architecture
Universitat Politècnica de Catalunya

Advisors: Joan-Manuel Parcerisa, Antonio González

March, 2020
Barcelona, Spain

Abstract

The computation capabilities of mobile GPUs have greatly evolved in the last generations, allowing real-time rendering of realistic scenes. However, the desire for processing even more complex environments clashes with the battery-operated nature of the devices integrating these kind of GPUs, such as smartphones and tablets, for which users expect long operating times per charge and a low-enough temperature to comfortably hold them. Consequently, improving the energy-efficiency of mobile GPUs is paramount to fulfill both performance and low-power goals. Previous works determined that the work of the processors from within the GPU and, notably, their accesses to off-chip memory are the main sources of energy consumption in graphics workloads. Yet most of this energy is spent in redundant computations, as the high frame rate required to produce smooth animations results in a sequence of extremely similar images.

The goal of this thesis is to improve the energy-efficiency of mobile GPUs by designing micro-architectural mechanisms that leverage frame coherence in order to reduce the redundant computations and memory accesses inherent in graphics applications.

Firstly, we focus on reducing redundant color computations. Mobile GPUs typically employ an architecture called Tile-Based Rendering, in which the screen is divided into multiple tiles that are independently rendered in on-chip buffers, thus reducing memory bandwidth. An analysis of popular Android applications reveals that it is common that more than 80% of the tiles produce exactly the same output between consecutive frames. We propose Rendering Elimination, a mechanism that accurately determines such occurrences by computing and storing signatures of the inputs of all the tiles in a frame. If the signatures of a tile across consecutive frames are the same, the colors computed in the preceding frame are reused, saving all computations and memory accesses associated to the rendering of the tile. Using commercial Android applications and state-of-the-art cycle-accurate simulators and models, we show that Rendering Elimination vastly outperforms related memoization schemes found in the literature, achieving a reduction of energy consumption of 37% and execution time of 33% with minimal overheads.

Next, we focus on reducing redundant color computations of fragments that will eventually not be visible. In real-time rendering, objects are processed in the order they are submitted to the GPU by the application, which usually causes that the results of previously-computed objects are overwritten by new objects that turn out to be closer to the observer and, therefore, occlude them. This phenomenon occurs because visibility is resolved on-the-fly along with the rendering process. Consequently, whether or not a particular object will be occluded is not known until the entire scene has been processed. Based on frame coherence and, therefore, the fact that visibility tends to remain constant across consecutive frames, we propose Early Visibility Resolution, a mechanism that predicts visibility based on information obtained in the preceding frame. Early Visibility

Resolution first computes and stores the depth of the farthest visible point after rendering each tile. Whenever a tile is rendered in the following frame, primitives that are farther from the observer than the stored depth are predicted to be occluded, and processed after the ones predicted to be visible. Additionally, this visibility prediction scheme is used to improve Rendering Elimination's equal tile detection capabilities by not adding primitives predicted to be occluded in the signature. With minor hardware costs, Early Visibility Resolution is shown to provide a reduction of energy consumption of 43% and execution time of 39%.

Finally, we focus on reducing computations in tiles with low spatial frequencies. Current GPUs produce pixel colors by sampling triangles once per pixel and performing color computations on each sampling location. However, an analysis of popular Android applications reveals that most regions of the screen do not include sufficient detail to require such high sampling rates, which leads to a significant amount of energy wasted computing the same color for neighboring pixels. Given that frame coherence implies that spatial frequencies are maintained across frames, we propose Dynamic Sampling Rate, a mechanism that analyzes the spatial frequencies of tiles once they have been rendered and determines the lowest sampling rate that maintains image quality, which is applied in the following frame. Results show that Dynamic Sampling Rate significantly reduces processor activity, yielding energy savings of 40% and speedups of 1.68x by only adding a small hardware unit to evaluate the spatial variations of a tile.

Acknowledgements

Haré todo lo que pueda y un poco más de lo que pueda, si es que eso es posible. Y haré todo lo posible e incluso lo imposible, si también lo imposible es posible.

– Mariano Rajoy

Even though I have read countless theses during the past few years, I have always smiled when stumbling upon an acknowledgements section and experiencing the joy left in a few paragraphs by a PhD candidate ecstatic to finish, dreaming of the day I would sit down to write such lines. Unbelievably, the moment has finally come to say:

I am extremely grateful to have had Professors Antonio González and Joan-Manuel Parcerisa as advisors. Thank you for everything you have taught me, for your motivation, for your guidance, for your patience and for the opportunity to have worked with you.

I will always hold a special place in my mind for Professor Ramon Canal for introducing me to the ARCO group and accompanying me in my first research steps and my first paper presentation.

I would also like to thank the colleagues with whom I have shared lab during these years. Thank you to the ones that graduated before me (Gem, Martí, Hamid and Reza) and showed me all the different lights that there are at the end of the tunnel. Thank you Jose Maria for the titanic task of having pioneered the graphics line of research in the group and thank you Enrique for being my unofficial mentor and providing me with all the help I could have wished for. Thanks to the colleagues that will graduate alongside me (Marc, Josué, Franyell and Albert) for having kept me sane by sharing the same journey of ideas that do not work, painful rejects and the ying-yang of research: coffee and beer. And a selfish thank you to the future of the group (Dennis, Raúl, Pedro, Jorge and Mehdi) for putting a mirror in front of me and involuntarily revealing me all the road I had already walked; I wish you a very successful PhD. Finally, I would like to thank Diya for having taken over the baton of graphics research and having let me test the waters in the advising world; I hope you manage to awe everyone with your work.

An endless thank you to the people that have cheered me and helped me evade from the PhD routine: thanks to the amazing Barcelona MtG community for infusing me with the desire to continuously thrive and discover, specially to my Nucli friends with whom I have shared victories, lessons and banters. And thanks to Alberto and Àlex for our eye-opening symposiums, which cemented the true purpose of research.

Thank you to my family for their unflinching support and care: none of this would have been

possible without your words of wisdom and esteem.

And the biggest thank you goes to the person who has done the most heavy lifting during these years and, extraordinarily, a little bit of everything listed above. Kiona, you alone have managed to turn the PhD years into the best years of my life. Now, let us enjoy the rest of the best years of our lives.

Contents

List of Figures

List of Tables

1 Introduction

This chapter presents the motivation behind the research interest in mobile GPUs and defines the challenges in their energy-efficient designs. Then, the specific issues that this thesis addresses are introduced, with an outline of the proposals to solve them and a comparison against state-of-the-art approaches. Finally, the main contributions of the thesis are listed.

1.1 The mobile Graphics Processing Unit: the driving force behind contemporary entertainment

Mobile devices have become essential for modern life [77]. They have greatly evolved in the past two decades and the so-called *Smartphones* are no longer tools for just calling or sending e-mails. Their broad connectivity, intuitive interfacing and remarkable computing capabilities allow for a wide variety of use cases, such as navigation or taking high-resolution pictures [21].

One of the paramount keys to the mobile revolution have been the *App Stores*, digital platforms to distribute third-party software to users [35]. These platforms tremendously streamlined the process of installing new programs, reducing it to a few taps in a screen. New applications were also extremely easy to discover, with clear browsing interfaces that sorted all the available offers into descriptive categories. Consequently, smartphones could add a myriad of functionalities, tailored to the individual needs of each user.

On the other side of the market, App Stores also provided advantages to developers, who experienced an abundant reduction in the overhead of releasing new applications: a centralized platform completely handled the connection with customers, software updates and, more importantly, monetization. Developers were rewarded for creating supply in a field full of user's demands (more than

100 billion dollars have been earned through Apple's App Store [7]), which led to a rapid increase in the number of available applications to download and purchase [78].

With an ever-increasing range of utilities, users began to perceive smartphones as tools that provided value in their daily lives. Soon, they were not only seen as useful but a necessity and their number of sales skyrocketed, becoming the technology that has been accepted the fastest in history [62]. Nowadays, smartphones have more than an 80% penetration rate in the majority of developed countries [19] and users prefer to carry out most of their ordinary activities on smartphone apps [18].

As societies have become accustomed to wear devices always connected to high-speed internet, their behavior has drastically changed, especially regarding entertainment [52]. In the smartphone era, immediacy and availability are favored above classical yearnings, such as product ownership. Consequently, streaming services are the way most of the media is consumed, from music to books or movies [55]. In particular, the ubiquity of smartphones has affected the video game industry the most.

The affordability and convenience of mobile devices in comparison with their console or PC counterparts have made them a video gaming phenomenon. As of 2019, a third of the world population plays games on their phones, spending more than 40% of their screen time on gaming applications [75]. The demographics have been hugely widened in the last few years and the traditional gender and age barriers have been blurred, with the average age of a mobile video gamer being 34 years old and the distribution between male and female users being nearly identical [76]. The video game market is extremely successful, responsible of the 76% of the overall app earnings and generating triple the revenue of the box office worldwide [74].

The industry has quickly adapted to these trends. On the one hand, software developers are continuously creating gaming mechanics that fit within what mobile technology provides. Smartphones have proven to be a very good interface thanks to their motion sensors and touch screens, which are a great way for interacting in dynamic and intuitive ways. Ease of control along with level designs that lend themselves to short bursts of gameplay have created the boom of the so-called *casual games* [44]: games without complex rules systems that can be played in the midst of day-to-day life, such as in the public transportation or in a break in the workplace. Developers have also managed to integrate geolocalization and cameras, smartphone-only features that were originally only functional, into completely revolutionary gaming experiences [61]. Nevertheless, studios have begun to successfully translate the traditional video game concept to mobile devices: rich scenes with detailed environments that can be interacted with intricate systems.

High-quality mobile graphics have become possible due to the incorporation of powerful Graphical Processing Units (GPUs). The release of the original iPhone in 2007 popularized a user interface design that required a dedicated graphics accelerator to enable its tactile direct manipulation interactions, such as pinch-to-zoom or inertial scrolling [36]. With smartphones integrating GPUs, a new era of mobile gaming started, where sophisticated applications could be rendered in real time. Semiconductor companies have consistently improved the capabilities of GPUs, achieving a performance evolution much faster than the one seen in previous decades with video game consoles. Figure 1.1 shows the performance in GFLOPS of the yearly releases of the iPhone, and compares them with the flagship consoles of Sony and Microsoft, the PlayStation and the Xbox. As can be

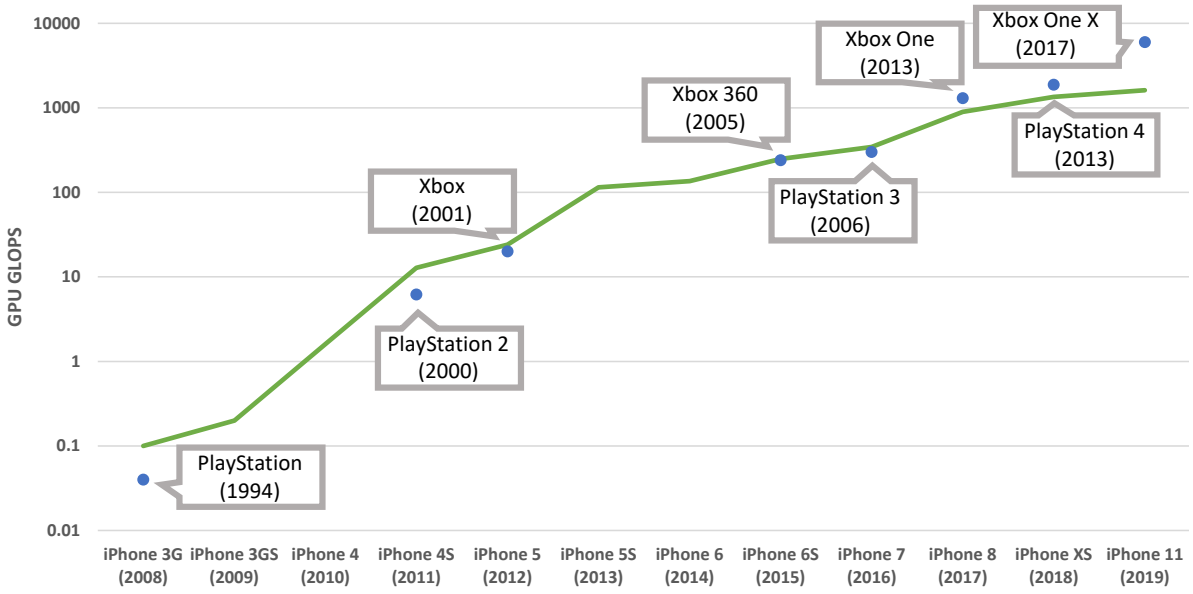


Figure 1.1: Performance evolution of the GPU in Apple's iPhone smartphone series compared to Sony's Playstation and Microsoft's Xbox console series. The release year of each product is displayed in parentheses below its name.

seen, smartphones have had the same GFLOPS increase in 10 years as video game consoles in 20. The performance gap between the two platforms is now closer than ever, with the iPhone 11 having an operation count similar to consoles at the start of the latest generation.

1.2 Problem statement

The Graphics Pipeline is the series of steps taken in real-time graphics to generate two-dimensional images given a three-dimensional scene. Figure 1.2 gives an overview of the pipeline. The geometry of the three-dimensional objects is modeled with a set of vertices, which are first processed, assembled into flat polygons (normally triangles) and projected into the screen plane. Triangles are then rasterized into fragments, pixel-sized regions of triangles, upon which a final color is computed and written in main memory to be displayed.

The aforementioned stages of the pipeline are accelerated in current systems by a GPU, the evolution of which has brought great improvements in visual quality. Increasing the performance of the hardware has allowed for more vertices to be processed, which translates into more detailed models. Additional computational power and memory bandwidth have been the basis for more complex shading models to apply in each fragment, which create realistic light and texture effects.

However, being able to sustain a pleasant experience comes at a great energy cost. Current mobile devices have adopted 1080p resolution, where screens are composed of 1920 pixels horizontally and 1080 pixels vertically. This implies that the color of more than 2 million fragments needs to be computed to render a single frame. Additionally, several tens of frames need to be displayed

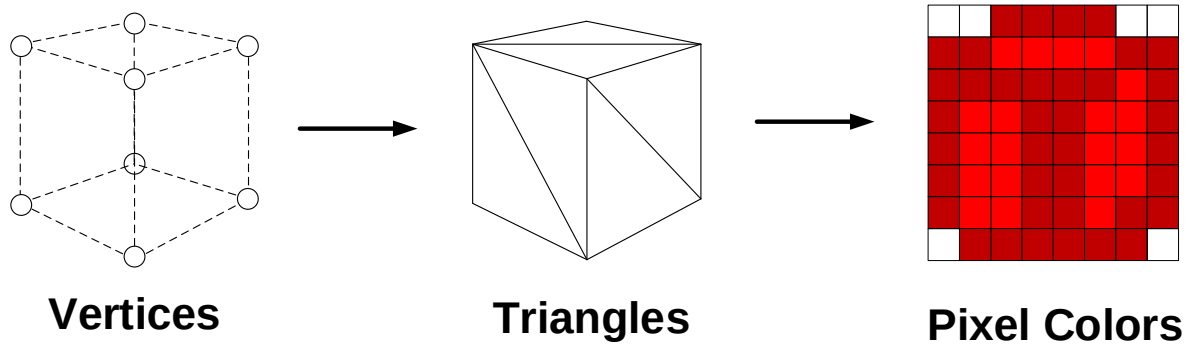


Figure 1.2: Graphics Pipeline overview.

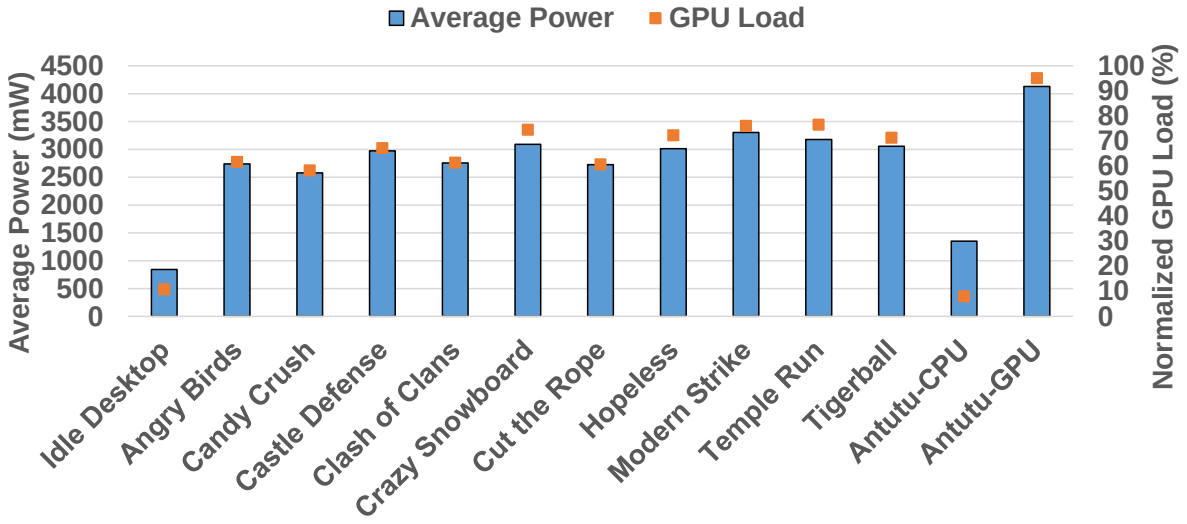


Figure 1.3: Overall average power consumption. GPU load is normalized by weighting it by the ratio between operating and maximum GPU frequency. Data obtained using Trepn Profiler [59] for a Snapdragon 636 with connections disabled (Wi-Fi and Cellular Data) and minimum screen brightness.

within a second for the human eye to perceive smooth motion. Nowadays, users expect games to produce fluid animations, for which a frequency of at least 60 frames per second (FPS) is required [51].

Figure 1.3 shows the average power consumption and GPU load for three types of applications. The first is the Android homescreen (*Idle Desktop*), the main screen of the most popular mobile operating system, consisting in stationary icons and widgets organized in a grid. Next, there is a series of smartphone games (*Angry Birds* to *Tigerball*), selected as representatives of the mobile landscape due to their large number of downloads in the App Stores and their game play diversity. The last application shown is the Antutu benchmark [6], a well-known benchmarking tool to rank smartphone performance in several categories, such as memory writing speed or multithreading. Two different component benchmarks are selected from the suite: the CPU test (*Antutu-CPU*), comprised of several tasks like FFT processing, and the GPU test (*Antutu-GPU*), which renders a 3D scene containing multiple real-rime lights. As it can be seen, even applications with simple

scenes such as the popular match-three game *CandyCrush* incur a substantial amount of GPU load, which drives an amount of power comparable to a benchmark designed to stress the GPU. Additionally, the amount of power that these games require is significantly higher than applications that leave the GPU mostly idle, illustrated by the Android homescreen, and is twice as much as an application that only stresses the CPU. Consequently, the design of energy-efficient GPUs is a priority for smartphones.

Energy consumption is paramount in battery-operated devices, as it dictates their autonomy. For instance, the Xiaomi Redmi Note 6 Pro, the smartphone used to obtain the numbers in Figure 1.3, has a battery capacity of 15Wh when supplied its typical voltage of 3.85V. If any of the listed games, whose average power lies between 2.5W and 3.5W, were to be played in this device, its battery would completely be drained in less than 5 hours.

While smartphones have seen a huge performance increase in the last generations, battery technology has not improved at the same pace. The consequence has been a continuously-increasing gap between the complexity of the scenes that can be rendered and the time they can operate without needing a battery charge. This has put a lot of pressure into energy-efficient mobile GPU designs in order to sustain the additional computational capabilities without affecting battery life, one of the most desired smartphone features by consumers [25]. Figure 1.4 shows the battery life and performance increase for the last installments of the popular line of Apple iPhone measured by GFXBench's *T-Rex* benchmark. In this test, a 56-second scene filled with modern effects such as soft shadows or planar reflections is looped 30 times. Performance is measured by logging the number of frames rendered in each loop and keeping the lowest one. On the other hand battery life is measured by logging battery discharge across loops and extrapolating the number of minutes it would take to completely drain the device's battery. The results of these tests show that, while graphics performance has increased by a factor of more than 6x in the last generations, the battery life of the devices when executing these type of applications has remained somewhat constant.

Reaching the desired FPS goal at a low power budget is not only a requirement for battery life, but also for temperature control. Packages have a thermal ceiling that cannot be surpassed before they are destroyed [40]. However, the design of smartphones impedes the application of traditional cooling mechanisms. They are made extremely thin and light to fit in pockets and be comfortably worn, so they cannot incorporate the characteristic large fans present in desktop graphic cards. Furthermore, they are devised to be held in the palm of a single hand, which greatly restricts the area available to dissipate heat. The thermal constraints are also much stronger in mobile devices than in other computing systems such as desktops because their typical use cases imply having them in contact with the body and it must be guaranteed that their surface temperature is not unpleasant [82].

Previous research has found the GPU and, in particular, fragment processing and its accesses to main memory, to be the greatest source of energy consumption when running graphics applications [3, 12, 54]. Figure 1.5 shows, using the simulation infrastructure described in Chapter 3, a detailed breakdown of the contribution of the different components of the GPU-Memory system to the overall energy consumption. The results match the literature statements by depicting communication with main memory, and notably the texture fetches generated by the fragment processing stage as the main reason for energy drain. From within the GPU, the fragment processors are the components that consume the most, as the fragment workload generally exceeds the vertex and triangle workload

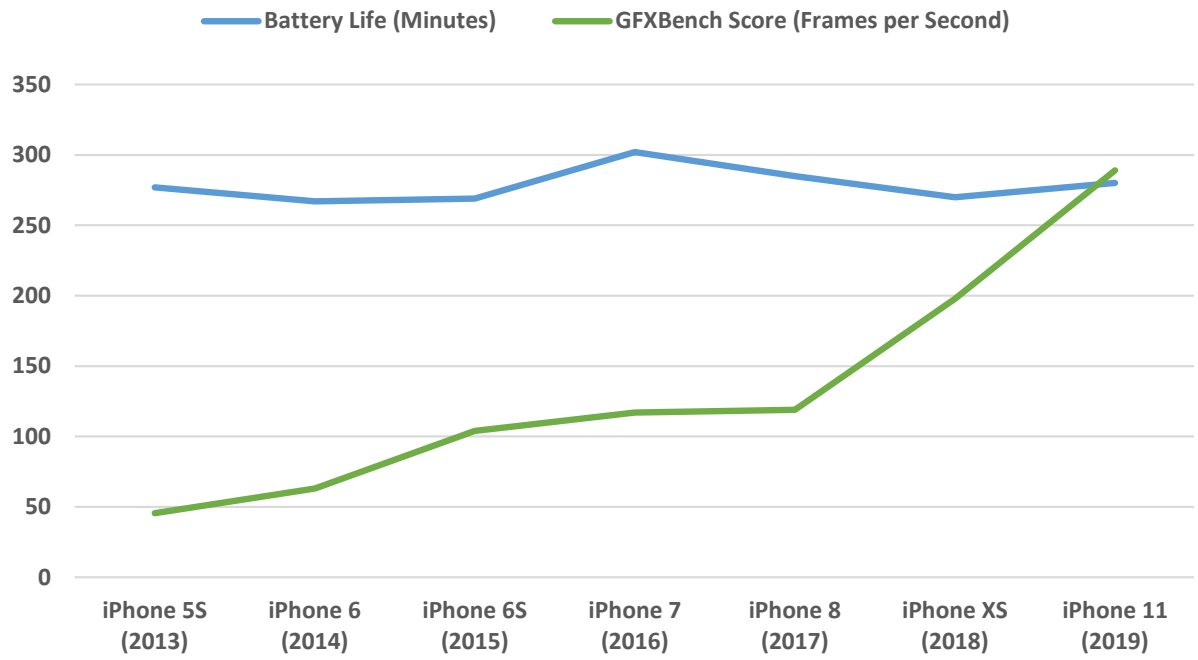


Figure 1.4: Battery life and GFXBench score evolution of Apple's iPhone smartphone series.

by more than two orders of magnitude.

However, most of this energy is spent in redundant operations. The high frame rate involved in creating smooth animations stems a succession of extremely similar images. This phenomenon, known as frame coherence, implies that a significant fraction of the inputs and outputs traversing the Graphics Pipeline in a particular frame are the same as in its preceding frame. In the next section, we outline three proposals to improve the energy efficiency of mobile GPUs by leveraging frame coherence to lessen redundancy in the most energy-consuming stage of the pipeline: fragment processing.

1.3 Thesis objective and related work

The objective of this thesis is to improve the energy efficiency of mobile GPUs by reducing the redundant computations and memory accesses inherent in graphics applications. The thesis presents three approaches that leverage frame coherence by implementing simple hardware designs that collect information in a frame to guide the execution of the following one in a way that the overall energy consumption is lowered. The three techniques are applied on top of Tile-Based Rendering, a common pipeline organization in mobile devices that divides the screen into rectangular sections -tiles- and renders them in succession, allowing the storage of temporary values in on-chip buffers and avoiding their corresponding accesses to main memory.

The following subsections describe inefficiencies in the pipeline of current mobile GPUs and present the proposals of this thesis to enhance energy efficiency and their novelty over the existing

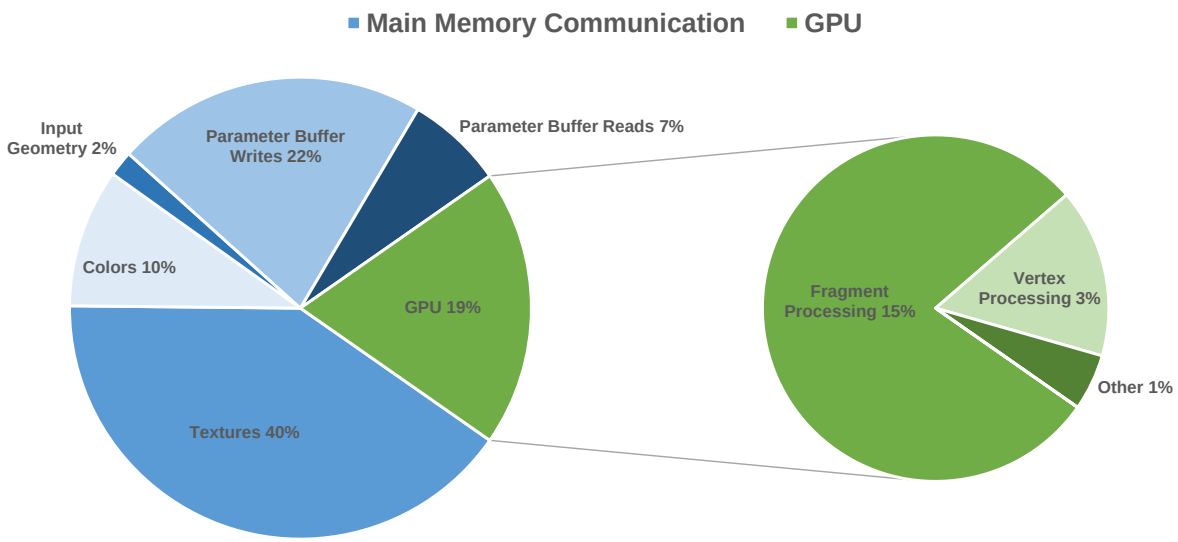


Figure 1.5: Average energy breakdown of a Tile-Based Rendering system executing the graphics applications listed in Chapter 3.

solutions in the literature.

1.3.1 Reducing redundant colors across frames

The Graphics Pipeline in a TBR GPU is divided into two decoupled pipelines [49]: the Geometry Pipeline receives vertices and generates, after a set of transformations, output primitives (triangles) that are sorted into tile bins and stored into the main memory Parameter Buffer. After all the primitives have been sorted, the Raster Pipeline processes the tiles one at a time, fetching each tile's primitives, rasterizing each primitive into fragments, and shading and texturing each fragment to obtain a final pixel color. Once all the primitives of a tile have been processed, its resulting colors are stored in the Frame Buffer, the main memory region from which the image is read to be displayed in the screen.

Frame coherence implies that the outcome of most tiles is exactly the same between two consecutive frames, which means that a significant portion of the operations devoted to render a frame are redundant. Several previous works have attempted to exploit frame coherence in order to remove these ineffectual computations and memory accesses to improve energy efficiency.

Transaction Elimination [41] (TE) is a bandwidth saving feature included in the ARM Mali GPUs that detects identical tiles between the current frame being rendered and a previously rendered frame. TE computes a Cyclic Redundancy Check signature of the output colors of a tile and compares it with the signature of the same tile in the preceding frame. If the signatures are equal, the tile is considered to be redundant and is not flushed to the Frame Buffer. As communication with main memory is the greatest source of energy consumption, avoiding the color flushes to main memory yields significant energy savings.

Parallel Frame Rendering [9] (PFR) is another technique to reduce communication with main memory, focused on texture bandwidth. Due to frame coherence, the same textures are likely to be reused across frames. However, due to the large texture dataset within a frame, most textures have been evicted from the caches by the time they are required again. PFR divides the Raster Pipeline into clusters which process consecutive frames in parallel with a fraction of the original GPU resources. Under this organization, a single texture fetch can be reused among all clusters, typically 2, greatly reducing the number of memory accesses.

Arnau et al. proposed to employ memoization [8] to avoid not only the texture accesses but the complete processing of redundant fragments. A small hardware lookup table stores the color results of fragments along with a hash of their input as an address. Subsequent fragments form their signatures and check them against the signatures of the memoized fragments. In case of a hit, the fragment skips its processing and the color is reused. Because most redundancy resides between consecutive frames, the huge reuse distance makes impractical to store a frame's worth of signatures and output values. To help reduce the reuse distance, the memoization scheme is build on top of PFR, which unfortunately cuts in half the redundancy detection potential: even frames reuse values cached by the previous (odd) frame, but odd frames cannot because their previous-frame values are already evicted from the LUT by the time they are rendered.

Hardware memoization has also been used in Decoupled Shading [39], a proposal to separate shading from visibility determination to reduce the cost of processing complex effects such as motion and defocus blur, which require sampling over a 5D domain (lens aperture and shutter interval besides the pixel area). Despite the large number of samples required to compute these effects, the colors of the samples within the same aperture or interval are very similar. In this approach, shading is sampled at a much lower rate than visibility by mapping visibility samples to shading ones using a function that accounts for blurring effects. Additionally, by memoizing previously-shaded results, the number of processed fragments is further reduced by reusing values for the same location. However, decoupling visibility and shading requires significant changes in a TBR architecture due to the synchronization point in the middle of the pipeline execution which essentially couples the visibility determination and shading of pixels within a tile.

We make the observation that in a TBR GPU, primitives do not need to be discretized into fragments to know that the final result will be the same as in the preceding frame. Instead, by managing redundancy at a tile level, redundant tiles may be discovered much earlier than at fragment level and bypass the whole Raster Pipeline. Note that the Raster Pipeline computes the pixel colors using as inputs a set of primitives' attributes generated by the binning stage of the Geometry Pipeline plus a set of scene constants, so it knows all the input data required to render a tile when it starts processing it.

Based on the above observation, we propose Rendering Elimination (RE), a novel technique that employs the input data of a tile to anticipate if all of its pixels will have the same color as in the preceding frame, and to bypass the complete rendering of the tile. Since an entire frame of these input sets must be stored on-chip, they are compared by means of a signature. In parallel with the sorting of a primitive into tiles, RE computes on-the-fly the signatures of the tiles it overlaps and stores them in a local fixed-size on-chip buffer. Then, after the Geometry Pipeline has processed the frame, tiles are dispatched to the Raster Pipeline. For each tile, RE compares its current and preceding frame signatures and, if they match, all the rendering process is bypassed and the colors

in the Frame Buffer are reused. Otherwise, the tile is rendered as usual.

By working at a much coarser grain than Fragment Memoization, RE can store on-chip all the frame signatures and detect all the available tile redundancy instead of just that of the even frames, which more than compensates for the marginal undetected redundancy at sub-tile level (our results show that RE almost doubles the amount of redundancy discovered). In addition, RE does not need to store output results because tile colors are reused from the Frame Buffer, thus saving storage and bandwidth. Besides this, while TE and Fragment Memoization each skip just a single stage of the Raster Pipeline (Color Buffer flush and Fragment Processing, respectively), RE completely skips all the Raster Pipeline stages. Additionally, unlike Decoupled Shading, implementing RE requires minimal changes to the common mobile graphics pipeline.

1.3.2 Reducing overshading

The Graphics Pipeline is responsible for resolving the visibility problem: determining which surfaces and parts of surfaces are not visible from a certain point. In current GPUs, 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. Fragments perform an *Early Depth Test* before they are shaded: their depth is compared with the one stored in their same position and are only processed and written to the Frame Buffer if they are closer than the previously visible fragment. However, as GPUs process vertices in the order that they are submitted by the application and do not perform any type of sorting, it is common for previously-computed fragments to be occluded by newer fragments that turn out to be closer to the observer. This phenomenon is known as *overshading*, and leads to an important energy waste as the color of pixels is uselessly computed multiple times.

We propose Early Visibility Resolution (EVR), a mechanism to reduce overshading based on a per-tile visibility estimation in early stages of the pipeline. Within a tile, occluded primitives are the ones whose depth is farther away from the viewpoint than the farthest visible point in that tile. By virtue of frame coherence, 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. EVR obtains the depth of the farthest visible points (FVP) of each tile in a frame after they have been rendered, and uses these depths to predict the visibility of primitives in the next frame. Whenever a primitive is binned into 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 predicted to be occluded for that tile, whereas if it is closer, the primitive is predicted to be visible.

We leverage this early visibility prediction scheme to reduce redundancy in a TBR Graphics Pipeline at two different granularities: First, at a fragment level, the effectiveness of the traditional Early Depth Test hidden fragment rejection is improved by processing primitives predicted to be occluded after those predicted to be visible. Second, at a 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. By construction of both Early Depth Test and Rendering Elimination, mispredicting the visibility of a primitive does not generate any errors: on the one hand, reordered primitives still have to perform the Depth Test, which maintains

correctness of the result. On the other hand, a visibility change requires a change in the attributes of one or more primitives, which results in different signatures between frames and, consequently, the rendering of a tile.

Hidden surface removal, and an efficient solution to it in particular, is a fundamental problem in computer graphics, with an extensive literature spanning more than three decades.

The Hierarchical Z Buffer [30] is a variation of the baseline Z Buffer technique in which a depth pyramid is used to test visibility. The base level of the pyramid corresponds to the Z Buffer 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 (the visibility after having shaded all the primitives in a frame) 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: the FVP abstraction allows EVR to also predict the visibility of primitives that do not use the Depth Test and instead are processed using the so-called Painter's Algorithm [46], where objects are processed in back-to-front order and displayed one over the other as if they were paint layers. EVR keeps track of the number of overlapping primitives within each tile that are processed in that manner using a counter named *Layer identifier*, which is used in the final FVP computation.

The concept of layers has been previously adopted in the context of occlusion culling, most notably in Depth Peeling [22], an algorithm that renders geometry multiple times. Each render pass processes only the fragments farther away from the closest depths in the previous pass and stores a new set of closest depths and colors, effectively peeling off the surface layer at each pass. After all the iterations have completed, all the generated layers are blended from back to front, guaranteeing correct transparency even for intersecting objects without the need to sort geometry. Recently, Andersson et al. [4] leveraged a two-layer representation of depths to avoid bandwidth spent in updating the Hierarchical Z Buffer. In a traditional pipeline with a Hierarchical Z Buffer, the coarser Depth Test is performed before rasterization to trivially accept or discard certain primitives, while the finer per-fragment Depth Test is performed before shading for all the non-trivial primitives. The results of the finer test may update one or more levels of the depth pyramid, which causes a feedback loop in the pipeline. This communication is undesirable because there may be a significant number of cycles of delay between the Hierarchical Buffer and the Depth Buffer, which increases the number of primitives passing the coarse test and, therefore, reducing its culling efficiency. In their approach, hierarchical tiles consist of one Z_{min} and a Z_{max} values per layer, which allows for depth information to be updated only during the coarse test and removing the feedback loop. In the work of Scheckel and Kolb [66], layers are used in combination with the alpha parameter to completely cull transparent fragments. It is common in 2D graphics applications to use the Painter's Algorithm to successively display objects stored in rectangular image buffers, which have their pixels completely transparent (their alpha component is set to 1) in areas outside the edges of the objects. This implies a significant energy waste in rasterizing and processing transparent fragments that do not contribute to the final color of the image. They propose to use min-max mipmaps to cull the completely transparent fragments. A pixel in a layer is completely opaque if

its min value is 1 and it is completely transparent if its max value is 0. Every 4 consecutive pixels within a layer are combined into one to generate the subsequent layer, and a hierarchical algorithm classifies layer areas as opaque or transparent by recursively checking for maximum and minimum pixel values. Starting from the coarser level, if a pixel represents a fully opaque rectangular region, the related layer is processed by generating two triangles. Otherwise, the algorithm traverses the hierarchy to the next finer level. Unlike EVR, these layer-based approaches cannot combine visibility information of both primitives that use the Depth Test and ones that do not for a better visibility determination.

Computing visibility at a fragment level (known as image-precision [71]) is useful to solve certain problems, such as circular dependencies. However, resolving visibility at a coarser grain could reduce the number of computations needed. Occlusion queries [67] are a feature supported by modern GPUs in which the application asks for the number of visible fragments of simple geometry (bounding volumes, for instance) and the hardware replies by testing it with the current contents of the Depth Buffer. With Coherent Hierarchical Culling [11], queries are scheduled using a hierarchical representation of the scene. The number of queries can be minimized by applying them in large regions expected to remain occluded. Additionally, the entire hierarchy is not completely traversed each frame: by taking advantage of temporal coherence, the traversal starts from the visible regions in the preceding frame, which are predicted to be visible in the current one. N-Buffers [17] are a different representation of the depth hierarchy which allows querying within a shader program for the depth of a rectangular region of arbitrary size and position in constant time. An N-buffer is a set of textures of identical resolution where a pixel p in the ith level corresponds to the maximum depth within a square area of 2^i x 2^i pixels around p. With this organization, it is possible to find the minimum depth in an arbitrary square area with only four texture accesses corresponding to four overlapping squares. While having a great potential to avoid the shading of occluded primitives, these queries involve a delay between issuing them and receiving the results, which greatly hinders performance. 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 axes: it requires neither synchronization nor ordering.

Several works perform additional auxiliary render passes to compute the depth of visible fragments and then execute the pass that shades them. Z-Prepass [14] is a software technique that draws the scene in two steps: first the geometry is rendered using null fragment shaders, that do not produce any color and are used 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. The Hidden Surface Removal stage in PowerVR architectures [58] is a hardware version of Z-Prepass for TBR architectures. The geometry is processed only once, in order to bin all the primitives in the scene into tiles. Each tile is then rendered using two steps: first, visibility is resolved by using simple shaders and obtaining the depths of the closest fragment in each pixel. The second step fetches and rasterizes all the primitives of the tile again, but the Early Depth Test is able to cull most of the workload sent to the Fragment Processors. Saito and Takahashi [64] introduced the G-Buffer, per-pixel intermediate storage for geometry information, such as normals or screen coordinates, used to enhance fragment shading. In particular, the G-Buffer stores depth information, so a first render pass fills the G-Buffer while removing hidden surfaces and a second pass reads the contents of the G-Buffer to compute the color of each pixel. 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.

Multiple works reduce overshading by means of reordering the primitives that make up a scene. Govindaraju et al. [29] propose an algorithm to sort objects in either front-to-back or back-to front order from a given viewpoint. The algorithm uses the Depth Buffer and the GPU to perform the image-space occlusion computations among objects. The work of Chen et al. [13] also creates a depth-sorted list for every possible viewpoint, focused on static objects that do not intersect. A preprocess creates back-facing duplicates for each input triangle in order to allow general triangle orderings. Cycles within the obtained occlusion graph are broken by using a heuristic that divides the space of viewpoints into a number of partitions and creating independent sorted lists for each partition. In the approach of Han and Sander [32], the preprocessing is extended to consider several key frames besides viewpoints, which generate sorted lists that can be indexed at runtime to reduce the overdraw in animated scenes. Weber and Stamminger [84] 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. Unlike EVR, these approaches are not suited for interactive scenes in real-time rendering due to their preprocessing needs, which is limited to certain camera angles, animations or frames. Visibility Rendering Order [15] was proposed recently as an approach to dynamically sort objects in real time. By taking advantage of temporal coherence, a visibility graph can be constructed entirely in hardware after rendering each frame and use it as an approximation for the visibility of the next one. Objects are thus sorted in front-to-back order and overshading is reduced by processing objects predicted to be occluded after objects predicted to be visible. EVR is able to reduce the overshading even further by working at a finer granularity (primitives instead of objects).

1.3.3 Reducing redundant colors within a frame

Primitives in the Raster Pipeline are discretized by sampling them across the screen and producing a fragment for each sampling location that they cover. Sampling points are usually placed in the center of pixels, which allows to capture the majority of details and avoid most aliasing effects like jagged edges or flickering. However, many regions of the screen do not contain enough level of detail to require such high sampling rates, leading to a significant amount of fragments wastefully producing the same color as their neighbours.

Based on that observation, we propose Dynamic Sampling Rate (DSR): a hardware mechanism that dynamically finds and applies, for each tile, the optimal sampling rate, i.e., the lowest sampling rate that does not cause visible artifacts in the rendered image. After the rendering of a tile to the on-chip color buffer finishes, DSR computes the 2D Discrete Cosine Transform of the resultant tile image, analyzes the spatial frequencies present in it and decides, based on a simple heuristic, the best sampling rate for the tile: whether it could have been sampled at a lower rate without sacrificing image quality, whether it contains enough detail that the sampling rate needs to be increased or whether the sampling rate is already optimal. Frame coherence implies that the level of detail of a tile across consecutive frames will be very similar, so the estimated-best rates for all the tiles are stored in a small on-chip Lookup Table and are queried during the rendering of each

tile in the following frame.

There is a lot of interest in the graphics community in sampling at coarser granularities than pixels because it is a direct way to reduce the number of shading executions, which entails reductions in execution time and energy consumption. DSR addresses the main shortcomings of prior work on this area.

Several techniques dynamically detect regions of the screen that can be sampled at lower rates by adding additional pipeline stages before or after the shading process. Deferred Adaptive Compute Shading [43] divides the framebuffer into levels, subsets of pixels progressively denser, and starts by shading the pixels in the coarsest level, the left corner in square regions of 4x4 pixels. The levels in the hierarchy are then traversed in order, processing all the pixels in a level before advancing to the subsequent level. Before shading a pixel, the values of the 4 closest, equidistant pixels from the previous level are compared using a user-defined criterion, such as threshold between color values combined with material identifiers. If the neighbor pixels are similar, the color of the pixel is computed by averaging their results and all the shading computations and memory accesses are avoided. Otherwise, the pixel is shaded as normal. Multi-Sampling Anti-Aliasing (MSAA) emulates the quality of sampling more than once per pixel without having to shade all the samples. Visibility is sampled many times per pixel, but the color computation is only performed once per pixel and shared to all the visibility samples covered by the triangle in that pixel. MSAA improves image quality especially around object silhouettes, where typically there are both color and depth discontinuities. In the work of Sathe and Akenine-Möller [65], MSAA is extended to reuse shading computations along internal edges from different primitives of the same object. They present a unit that queues fragments before the processing stage and detects non-overlapping fragments from neighbouring primitives using their coverage information and vertex identifiers. In Adaptive Image-Space Sampling [69], the resolution is reduced in areas that contain less perceivable detail. Using the information available after the geometry processing, a visual perception method based on the human visual system evaluates the probability of a fragment being important to the final image. Only the fragments labelled as necessary are shaded, while the color of the others is obtained by interpolating other results. Unlike these approaches, DSR is architected to not introduce any time overheads by completely overlapping the sampling rate estimation for a tile with the rendering of the next one.

To avoid the runtime overhead of determining components with less detail, several works allow the programmer to statically determine the sampling rate. In Coarse Pixel Shading [79], the sample rate of a primitive can be controlled based on their vertex attributes. The sample rate may change for a primitive across different regions of the screen and different attributes may also be sampled at independent rates. He et al. [33] design new language abstractions that grant each shader program the ability to sample different components of the shading function at different rates. The rasterizer first generates coarse fragments, which execute their shader for effects that have low spatial variation, like certain lights. These fragments are then partitioned into finer fragments, which execute traditional shader programs for effects that have per-pixel variability. In the new Turing GPUs, NVIDIA has introduced Variable Rate Shading, a feature that allows the programmer to decide which sampling rate to apply in each 16x16-pixel region of the screen. DSR, on the other hand is able to continuously adapt to changes in the scene by dynamically estimating the best sampling rates in each tile using a hardware-only mechanism in a completely transparent manner

to the programmer.

Frame coherence has been previously leveraged to reduce the number of samples to process. In Checkerboard rendering [47], each frame shades an alternate half of the pixels in the screen. The color of the non-shaded half is obtained by applying reconstruction filters to the results obtained in the preceding frame. A large number of shading computations are avoided at the cost of some visual artifacts, since the lossy nature of the reconstruction and the fixed undersampling cannot perfectly reproduce neither motion nor visibility changes. In contrast, DSR estimates sampling rates at the finer granularity of tiles, can render tiles at the small rate of only one fragment per tile and does not affect image quality because it only reduces the sampling rate whenever a tile does not contain high spatial frequencies.

1.4 Thesis contributions

This section presents the contributions and publications of this thesis' research on energy efficient mobile GPUs for graphics applications.

1.4.1 Rendering Elimination

Rendering Elimination (RE) is a coarse-grained memoization scheme that avoids the entire processing of redundant tiles. The work has been published as:

Rendering Elimination: Early Discard of Redundant Tiles in the Graphics Pipeline. Martí Anglada, Enrique de Lucas, Joan-Manuel Parcerisa, Juan Luis Aragón, Pedro Marcuello and Antonio González. In 2019 IEEE International Symposium on High Performance Computer Architecture (HPCA'19).

Its main contributions can be summarized as follows:

- The observation that frame redundancy can be discovered in a Tile-Based Rendering GPU at the tile level much earlier in the pipeline than previous techniques do.
- An analysis of the large amount of tile-level redundancy in current graphics applications, which leads to energy waste when computing again the same colors for a tile than in its preceding frame.
- A detailed proposal of a mechanism to detect tile redundancy in early stages of the Graphics Pipeline and avoid its processing.
- An architectural implementation of the tile redundancy detection that can be integrated into the Graphics Pipeline with minimal hardware and performance overheads.
- An experimental evaluation of RE that shows that our proposal to discard redundant tiles yields energy savings of 37% and an execution time reduction of 33% over a conventional mobile GPU, and substantial improvements over previous works.

1.4.2 Early Visibility Resolution

Early Visibility Resolution (EVR) is a mechanism to avoid the processing of fragments belonging to hidden surfaces. The work has been published as:

Early Visibility Resolution for Removing Ineffectual Computations in the Graphics Pipeline. Martí Anglada, Enrique de Lucas, Joan-Manuel Parcerisa, Juan Luis Aragón, and Antonio González. In 2019 IEEE International Symposium on High Performance Computer Architecture (HPCA'19).

Its main contributions can be summarized as follows:

- An analysis of the amount of overshading in current graphics applications, which leads to energy waste when computing the colors of a primitive that will eventually be occluded.
- A detailed proposal for a mechanism to estimate visibility at tile level in early stages of the Graphics Pipeline based on exploiting frame coherence.
- A mechanism to employ the visibility determination to reduce overshading by processing primitives predicted to be visible before primitives predicted to be occluded.
- A mechanism to employ the visibility determination to improve Rendering Elimination's redundant tile detection by not including primitives predicted to be occluded in the memoization scheme.
- An architectural implementation of a visibility prediction scheme that can be integrated into the Graphics Pipeline with minimal hardware overhead and without any performance penalty.
- An experimental evaluation of EVR that shows great reduction in overshading and improvements in redundant tile detection, yielding energy savings of 43% and an execution time reduction of 39%.

1.4.3 Dynamic Sampling Rate

Dynamic Sampling Rate (DSR) is a technique that reduces the number of fragments generated in tiles with low spatial frequencies. The work is currently under review as:

Dynamic Sampling Rate: Harnessing Frame Coherence in Graphics Applications for Energy-Efficient GPUs. Martí Anglada, Enrique de Lucas, Joan-Manuel Parcerisa, Juan Luis Aragón, and Antonio González.

Its main contributions can be summarized as follows:

- An analysis of the number of tiles in current graphics applications that can be sampled at a lower rate than the baseline without producing visual artifacts, which leads to energy waste when processing the colors of fragments that do not improve image quality.

- A new hardware technique, completely transparent to the programmer, that estimates the lowest possible sample rate to which each tile may be rendered without producing visual artifacts and applies it during the following frame by taking advantage of frame coherence.
- A dynamic mechanism based on the real-time analysis of the spatial frequencies, that continuously adapts the sample rate of each tile to track the image changes that occur over time.
- An architectural implementation of the frequency analysis unit that can be integrated into the Graphics Pipeline with minimal hardware overhead and without any performance penalty.
- An experimental evaluation of DSR that shows great reduction in redundant shader activity, yielding energy savings of 40% and an execution time reduction of 36%.

2

Background: Tile-Based Rendering

This chapter provides a brief overview on the Graphics Pipeline. While the Pipeline is a conceptual model that can be implemented in many ways, the focus of this chapter is set on how data is transformed and transported across the different stages in Tile-Based Rendering, an architecture designed with low power as a goal. The objective of the chapter is to give background on the tasks that the GPU solves and introduce the terminology that will be used throughout the document, in particular of Tile-Based Rendering, as it is the design that will be used as a baseline in the proposals of the thesis.

The Graphics Pipeline is the process that obtains a two-dimensional image given a three-dimensional scene and a camera position and orientation. Three-dimensional scenes usually consist of light sources and three-dimensional objects, which are modelled by polygons. Rendering is a complex operation consisting of many operations and is, consequently, pipelined to improve its performance and throughput. All the required steps can be grouped in three coarse pipeline stages where each stage requires the output of the previous one as its input: Application, Geometry and Raster, as seen in Figure 2.1. The Application Stage corresponds to the program running on the CPU, and is responsible to create the geometry that defines the scene, to adapt it according to the user interaction and to send it to the GPU. The GPU implements the next two stages with a mix of fixed-function and programmable hardware: the Geometry Stage, in which the geometry is transformed and assembled into triangles in the screen plane, and the Raster Stage, in which triangles are discretized into "picture elements" (*pixels*) that are then shaded: a color is computed for each of them.

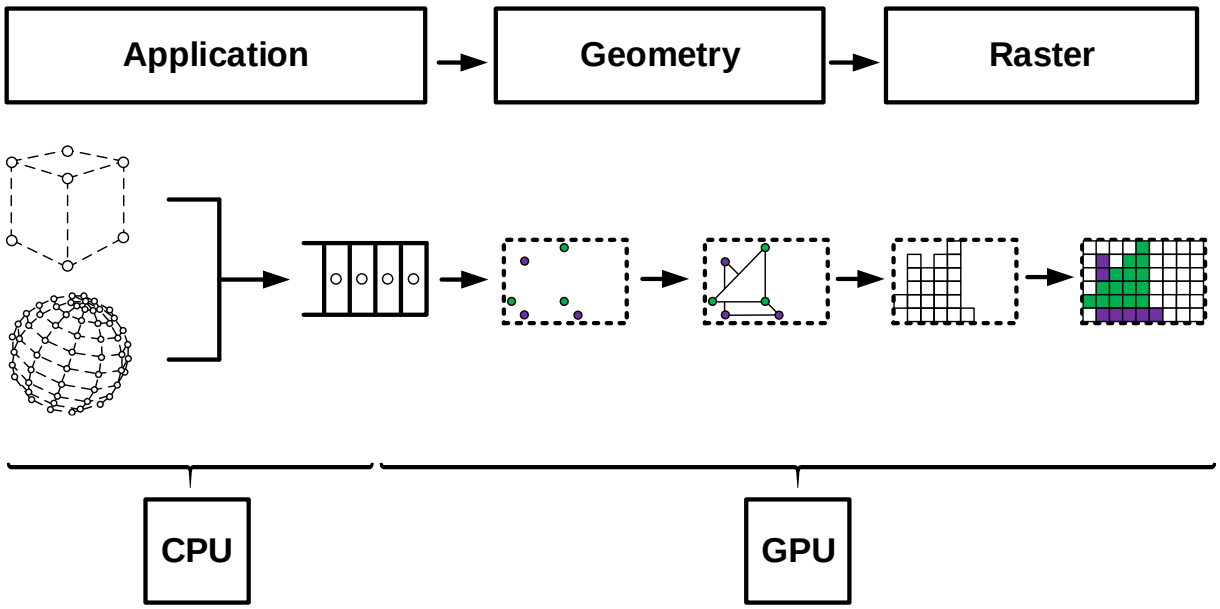


Figure 2.1: Coarse view of the Graphics Pipeline.

2.1 The Application Stage

The Application Stage is the collection of software instructions required to set the state of the pipeline. This encompasses a plethora of operations, ones that manage interfacing and ones closer to managing the hardware.

On the high-level side, the application controls how the scene reacts to the user inputs, establishing the motion of the objects and the camera and deciding which objects should be displayed. The programmer also determines how objects interact with each other and the world by computing the effect of forces upon them, particularly collisions.

On the low-level side, a wide variety of capabilities and optimizations can be enabled, such as culling, depth test or blending, which will be discussed later in this chapter. The application also manages memory resources and how data is stored and handled. The most important function of the Application Stage is to load the objects belonging to the scene and transfer them to the GPU. Objects are comprised of vertices, points in three-dimensional space to which the application associates additional information known as *attributes*, such as normals or color. Attached to the objects are also *shaders*, code defined by the application that computes how different types of light interact with their surfaces, and *textures*, images that can be applied on top of polygons to add high-frequency detail. The application communicates with the GPU the state of the pipeline for a batch of vertices using *Commands* and triggers them using a *Drawcall*.

2.2 The Geometry Stage

Figure 2.2 shows a block diagram of the Geometry Stage, which receives input streams of vertices from the GPU and transforms it into a series of 2D primitives.

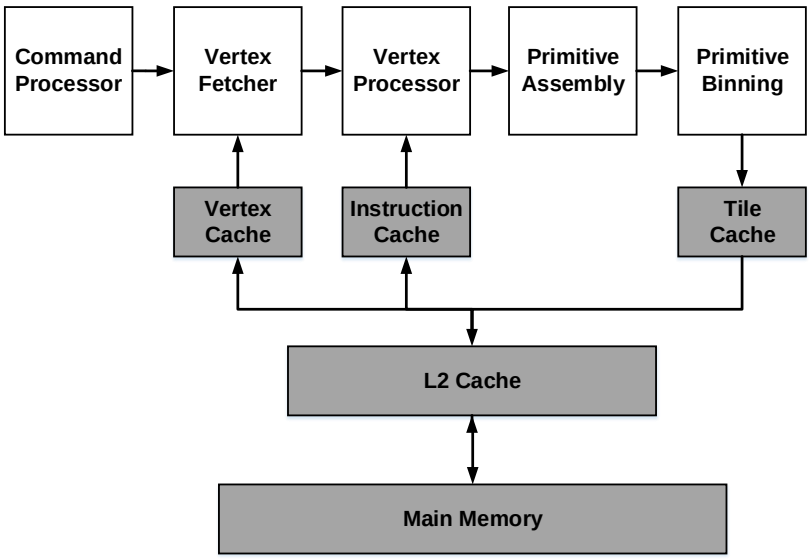


Figure 2.2: Geometry Pipeline

Command Processor

The Command Processor reads the commands sent by the application and configures the different stages of the GPU. In particular, it parses how the vertex information of each Drawcall must be interpreted: how do vertices form primitives and how their attributes are laid out in memory. The ordered list of vertices that the GPU receives can represent a wide variety of primitives, depending on the application specification. Figure 2.3 shows the different possibilities in which vertices can be connected. The most used primitive types are triangles, in which every group of 3 vertices forms a triangle, and triangle strips, in which successive vertices form triangles with the preceding two.

Vertex Fetcher

The Vertex Fetcher reads the vertices requested by a Drawcall and unpacks its attributes to be used as inputs in the vertex shader. As illustrated in Figure 2.4, vertices in three-dimensional objects tend to be shared by several triangles. Consequently, the Vertex Fetcher is aided by a Vertex Cache to capture that reuse and reduce vertex communication with main memory.

Vertex Processor

Vertices fetched from memory are transformed by executing user-defined shader code in the Vertex Processor. The main purpose of the vertex shader is to compute the 2D coordinate in the screen corresponding to a vertex 3D position. The final location of a vertex is obtained by performing a series of affine transformations to its position attribute:

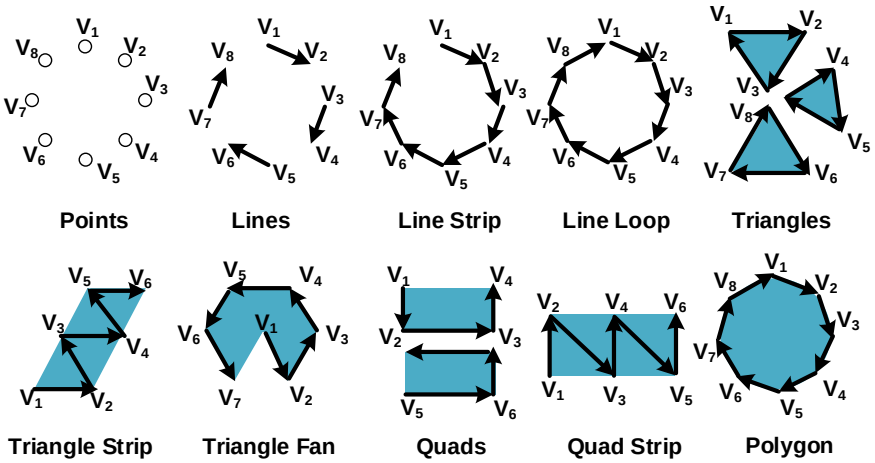


Figure 2.3: Examples of primitives represented by a vertex stream. The subindex in each vertex corresponds to its submission order.

- **Model Transform.** Each object is initially placed in its own coordinate space, with the position of its vertices relative to an individual origin. The model transform positions vertices in a common coordinate system called *world space*.

- **View Transform.** Vertices are placed into *view space*, an auxiliary coordinate system where the camera is at the origin looking at the negative Z-axis. The view transform orients the world around the camera position to facilitate projection computations.

- **Projection Transform.** Vertices in the camera's viewing volume, defined by the camera's field of view and by a near and far planes, are projected to the screen plane. The Projection Transform defines the mapping between the 3D space and the 2D plane and yields coordinates in *clip space*, a system useful for clipping, the next step in the pipeline that discards primitives outside the viewing volume. The perspective projection, in which objects farther away from the camera appear smaller and parallel lines converge to a single point, is the most widely used projection.

These operations are implemented using transformation matrices in the homogeneous coordinate system, where a 3D location (x,y,z) is represented as [x,y,z,w=1] and a 3D direction (x,y,z) is represented as [x,y,z,w=0]. By representing vertices in homogeneous coordinates, all the aforementioned transforms can be computed using a concatenation of 4x4 matrix multiplications, where coordinates can be rotated, translated or scaled as shown in Figure 2.5.

The Vertex Processor is thus architected to quickly perform these transforms. It consists of a simple, in-order pipeline composed by Fetch, Decode, Execute and Writeback stages built around 4-wide vector floating point units to compute results for each [x,y,z,w] component. The Single Instruction Multiple Thread (SIMT) execution model is employed to tolerate latency and provide high throughput: multiple threads are bundled in a warp and progress in lock-step by executing the same shader instruction on different vertices, and several warps are running concurrently by interleaving their execution. SIMT is a great fit for vertex workloads, as the same shader code is

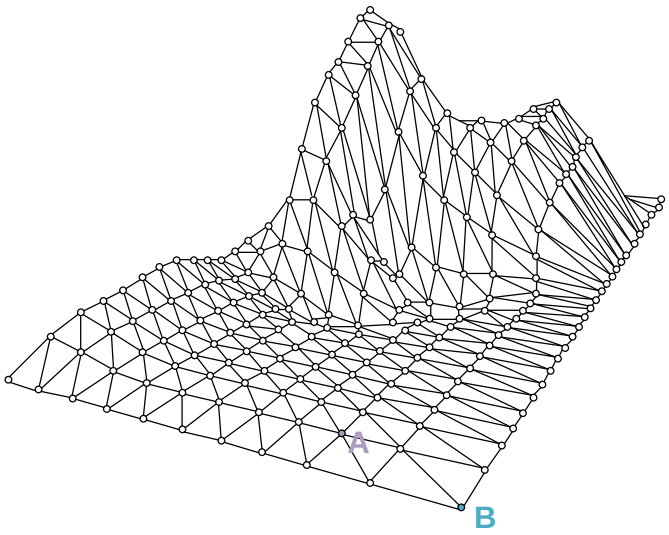


Figure 2.4: 3D Model of a mountain hill. Vertex A is shared by 6 triangles. Vertex B is shared by 2 triangles.

shared with a large amount of vertices and shader invocations are independent: they do not need information from other vertices and, therefore, their execution can be massively parallelized.

Shader code is written using C-like languages with their own programming model, and is compiled to the specific ISA of the GPU by a driver. In essence, shaders produce outputs to be consumed later in the pipeline using two types of inputs: values that change per shader invocation (such as vertex attributes) and *uniforms*, values that remain constant throughout a Drawcall. In particular, vertex shaders are required to output at least a 2D position, but they can also modify other attributes that model the appearance of objects, such as color, normal or texture coordinates. The number of registers used by a shader determines the maximum number of simultaneous warps that can be executing concurrently, as the interleaving of warps is possible due to a big shared register file that stores the state of all in-flight threads.

Primitive Assembly

The Primitive Assembly groups the shaded vertices that leave the Vertex Processors into the primitives that the application used to define the geometry (Figure 2.3). Additionally, it performs two visibility-related optimizations, clipping and culling, and a last transform to position vertices into coordinates in the screen window. A visual summary of all the vertex transforms performed in the Geometry Pipeline is shown in figure 2.6.

- **Clipping.** Primitives that lie fully outside the viewing volume are removed from further processing, as they will not appear in the final image. For primitives that lie partially inside the viewing volume, the portion that will not be visible is discarded by replacing vertices located outside the viewing volume by vertices located at the intersection between the edge of the primitive and the planes of the viewing volume. As shown in Figure 2.7, the clipping process for these primitives discards some vertices, creates new ones and forms new primitives. Primitives that lie fully inside the viewing volume are sent to the next step in the pipeline

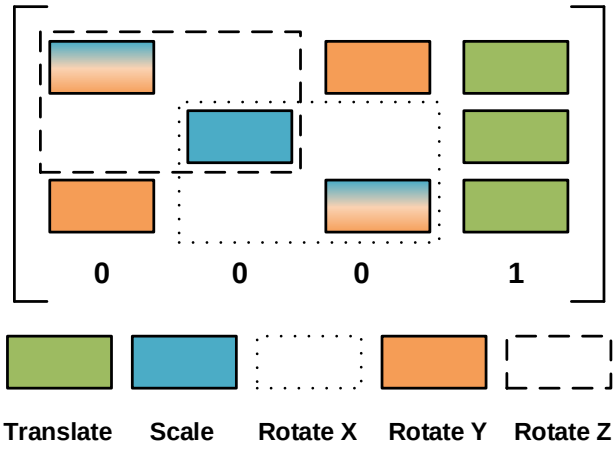


Figure 2.5: Matrix transform.

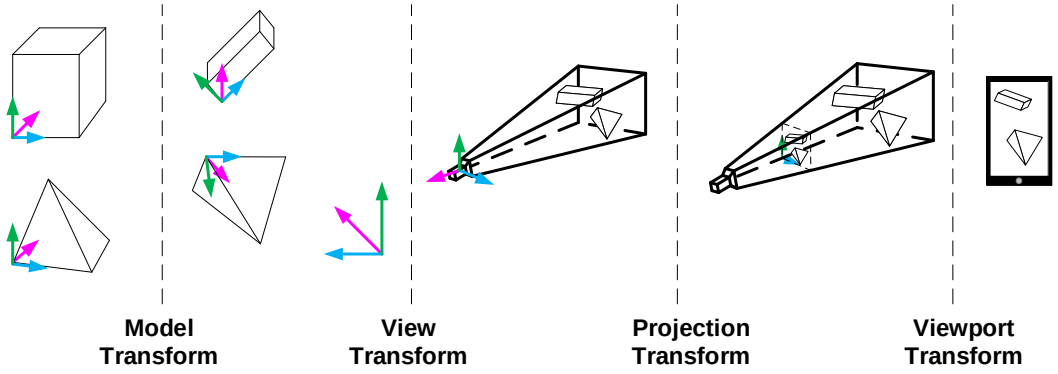


Figure 2.6: Vertex transforms.

without any processing. Vertex coordinates in clip space can be easily tested to be located inside the viewing volume by checking if its (x,y,z) coordinates are within the range [-w, w].

- **Viewport Transform.** After the projection, an additional step called *perspective divide* transforms coordinates in the clip space into *normalized device coordinates* by dividing their (x,y,z) components by their w value. This results in (x,y,z) values in the (-1,1) range, which can be easily scaled and translated by a final transform that maps vertices to actual pixel coordinates taking into account the resolution of the display.

- **Backface Culling.** Triangles that face away from the camera are removed from further processing, as the triangles that face towards the camera will occlude them. Detection of backfacing triangles is done by analyzing the winding order of the vertices. Whenever triangles are submitted to the pipeline, their front face is defined by sending their vertices in a specific order, either clockwise or counter-clockwise, as shown in Figure 2.8a. After triangles are in normalized device coordinates, if the winding order of a triangle is the same as the one that was submitted, it is front-facing. Otherwise, it is a back-facing triangle and it is culled (Figure 2.8b).

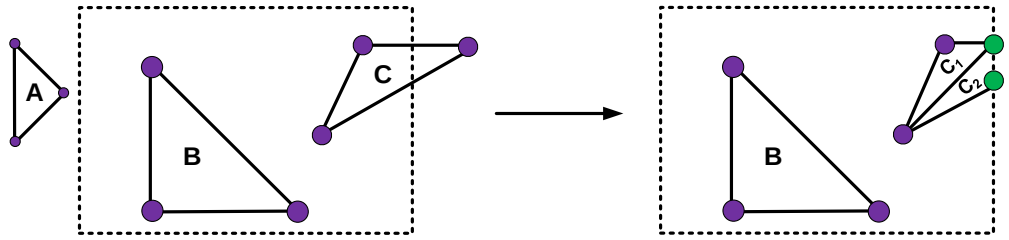
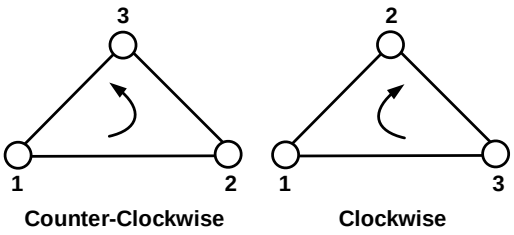
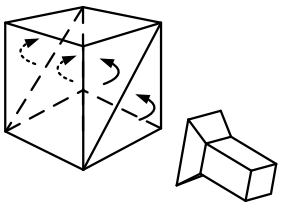


Figure 2.7: Primitive A is clipped because it is completely outside the viewing volume. No change occurs to primitive B, as it is completely inside the viewing volume. Primitive C is partially outside, so new vertices are created, forming triangles C_1 and C_2.



(a) Winding orders for a primitive, defining its front-face.



(b) Primitives whose front-face looks toward the camera have their vertices in counter-clockwise order, while primitives whose back-face looks towards the camera have their vertices in clockwise order.

Figure 2.8: Backface determination.

Primitive Binning

The traditional desktop GPU architecture is commonly known as *Immediate Mode Rendering*, where primitives directly advance to the Raster Pipeline after being assembled. Immediate Mode Rendering requires a lot of communication with main memory, as triangles in the stream could be positioned anywhere in the screen and, consequently, the working set for temporary values is too large to be conveniently cached.

Mobile GPUs usually employ another approach called *Tile-Based Rendering*, where the screen space is divided into a regular grid of *tiles* which are independently processed. By rendering only a small region of the scene at a time, a variety of computations in the Raster Stage can leverage local on-chip memories for temporary results instead of using main memory. In Tile-Based Rendering, the results of the Raster Stage (the colors of the pixels composing a tile) are only written to main memory once, after the rendering of a tile is complete. Therefore, all the primitives that could contribute to the colors of a tile must be known before starting its processing.

The Geometry and Raster Stages are decoupled by the Primitive Binning, a step that generates a data structure that indicates which primitives overlap in each screen tile. Additionally, the output of the Geometry Stage (the per-vertex transformed attributes of primitives) must also be stored so that the Raster Stage can later operate on it after all the primitives in the scene have been binned. This data structure is known as *Parameter Buffer*, and is stored in main memory. Accesses to the Parameter Buffer are aided with a Level-1 cache called Tile Cache, as primitives assembled consecutively tend to overlap a similar set of tiles.

Tile-Based Rendering saves memory accesses during the Raster Stage at the expense of generating additional geometry-related memory accesses. It is generally a good trade-off in terms of energy consumption, as the Raster workload is several times larger than the Geometry workload. However, the decoupling process introduces a significant performance hit that high-end graphics applications cannot tolerate. Thus, Tile-Based Rendering is normally confined to power-constrained GPUs.

2.3 The Raster Stage

Figure 2.9 shows a block diagram of the Raster Stage, which processes the screen one tile at a time, identifying the pixels inside each primitive and determining their colors.

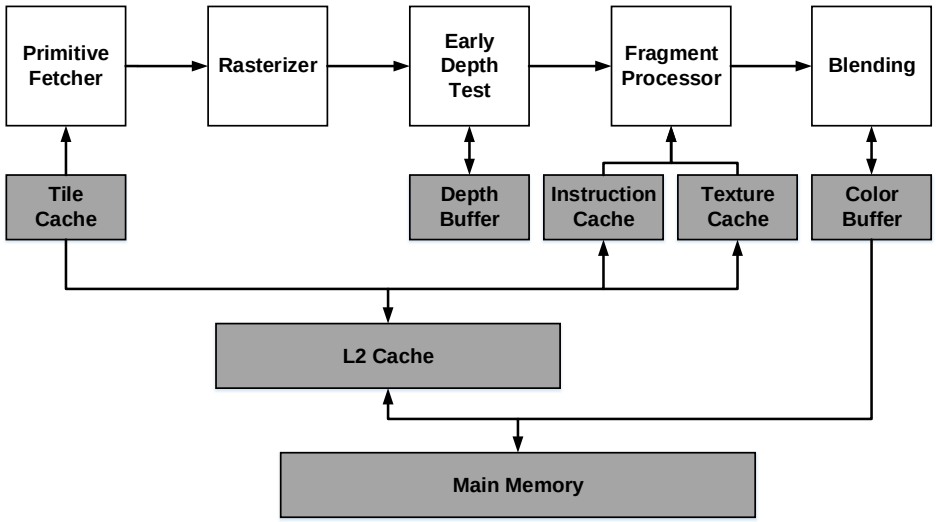


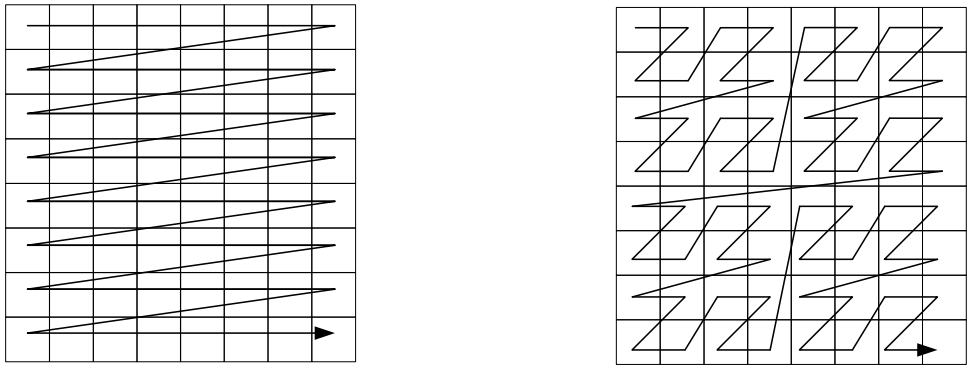
Figure 2.9: Raster Pipeline.

Primitive Fetcher

The Primitive Fetcher retrieves the primitives of a tile, stored in the Parameter Buffer. Accesses to the Parameter Buffer are aided with a Tile Cache that allows the efficient reuse of fetched attributes, as primitives tend to overlap several tiles close together in the screen. In order to maximize the number of cache hits, tiles in the screen are not processed in scanline order (i.e., row-major order) and are instead rendered using traversals that increase locality, such as the z order curve [50] (Figure 2.10).

Rasterizer

Rasterization is the process to determine which pixels in the screen are covered by each primitive. In current GPUs, this is done by performing what is known as *inside-outside test*, which checks whether the point in the center of a pixel is inside or outside the primitive. If that is the case, a *fragment* is generated for that position: a set of data necessary to generate the color for a pixel, obtained by interpolating the attributes of the primitive's vertices in the location of the fragment. The Rasterizaton of a primitive is, therefore, divided into two steps: Triangle traversal (finding



(a) Scanline *(b) Z Order*

Figure 2.10: Possible tile traversal orders.

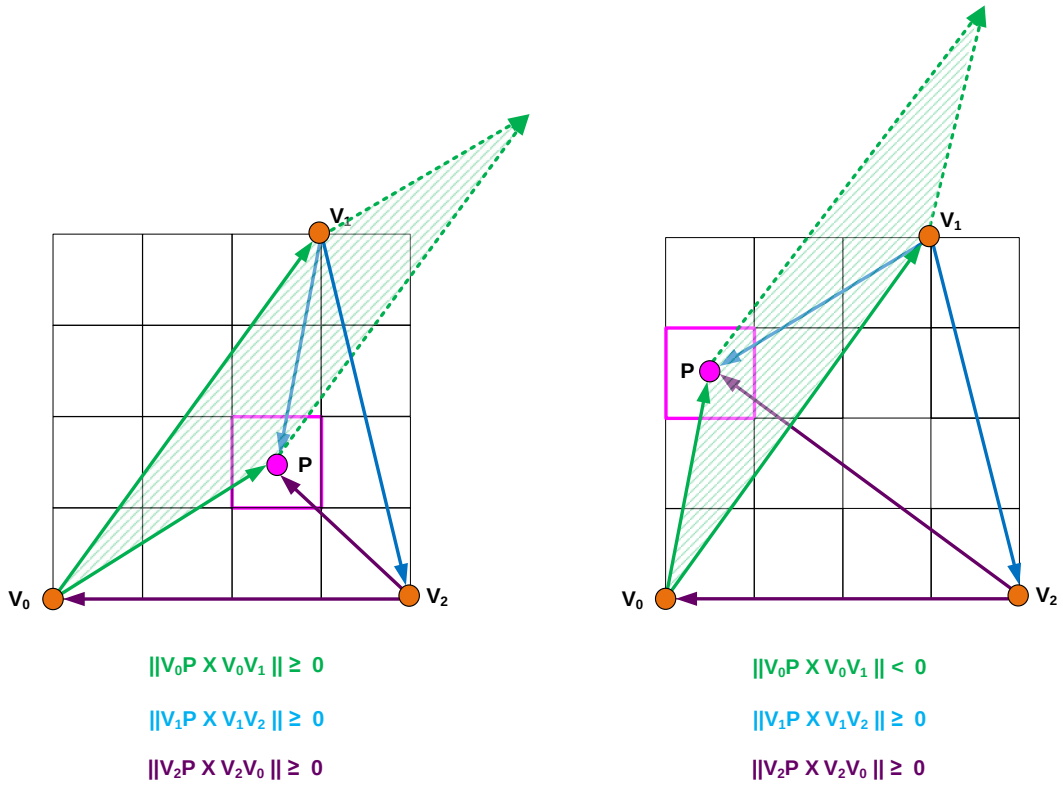
which pixels overlap the primitive, which is in the majority of cases a triangle) and Interpolation (generating fragments to obtain the colors of the primitive).

- **Triangle Traversal.** Edge functions [57] are used to determine whether a point (generally the center of a pixel) is inside a triangle. An edge function classifies points in a 2D plane subdivided by a line into three regions: points to the "left" of the line (the function returns a negative value), points to the "right" of the line (the function returns a positive value) and points on the edge (the function returns 0). The three edges of a triangle formed by vertices v_0, v_1 and v_2 are the vectors $(\overrightarrow{v_0v_1})$, $(\overrightarrow{v_1v_2})$ and $(\overrightarrow{v_2v_0})$. If the edge function for the center of a pixel (P) returns a positive value for the three edges, the pixel overlaps the triangle. The edge function is linear, which has the implication that can be incrementally computed for a pixel by adding the position displacement to a previously computed edge function for another pixel. As position displacements are independent, edge functions can be computed in parallel for several pixels at once if the edge equations for a reference point have previously been computed, a step known as *Triangle setup*.

  The edge function is computed using the sign of the cross product of an edge and another vector defined by P and the first vertex of the edge (Figure 2.11). The cross product can be interpreted as the signed area of the parallelogram formed by the two vectors, where the sign of the area indicates the orientation of the vectors with respect to each other and has the same behavior as the edge function.

- **Interpolation.** Barycentric coordinates are used to weight the contribution of each vertex in a primitive to a particular location. A point P inside a triangle whose vertices are v_0, v_1 and v_2 can be defined with the barycentric coordinates w_0, w_1 and w_2 as $P = w_0 * v_0 + w_1 * v_1 + w_2 * v_2$, with $w_0 + w_1 + w_2 = 1$. With barycentric coordinates, any vertex attribute across the surface of the triangle can be easily interpolated. The Barycentric coordinates of P with respect to the triangle $v_0v_1v_2$ are equivalent to the ratios of the area of the subtriangles Pv_1v_2, Pv_2v_0 and Pv_0v_1 with respect to the area of the base triangle $v_0v_1v_2$ (Figure 2.12). The area of the triangle is computed only once during Triangle setup, as it is constant for the computation of the coordinates of all its pixels.

  The area of each subtriangle corresponds to half of the area of the parallelogram formed by



(a) P is inside the triangle because the 3 areas formed are positive.

(b) P is outside the triangle because the area w.r.t. V0 is negative.

Figure 2.11: Edge equations example.

an edge of the base triangle and the vector defined by P and the first vertex of the edge. Therefore, barycentric coordinates are computed using half the value of the cross product between the two vectors. A fragment is generated by computing the barycentric coordinates for the center of a pixel and interpolating all vertex attributes.

Early Depth Test

The final image of a rendered scene only contains the visible primitives, that is, the color of each pixel corresponds to the closest fragment from the camera's point of view. Visibility in mobile graphics applications is handled in one of two ways: the Painter's Algorithm or the Depth Test.

The Painter's Algorithm draws the scene from back to front, and newly-processed opaque fragments in a position of the screen always occlude previously-rendered fragments in that position. It requires that the application sorts all the objects in the scene so that they can be processed in this order, a task that sometimes may require a lot of time due to the large amount of objects needed to be sorted or that sometimes, as in the case of intersecting objects, may not even be possible.

Therefore, visibility of overlapping fragments is typically and more conveniently handled em-

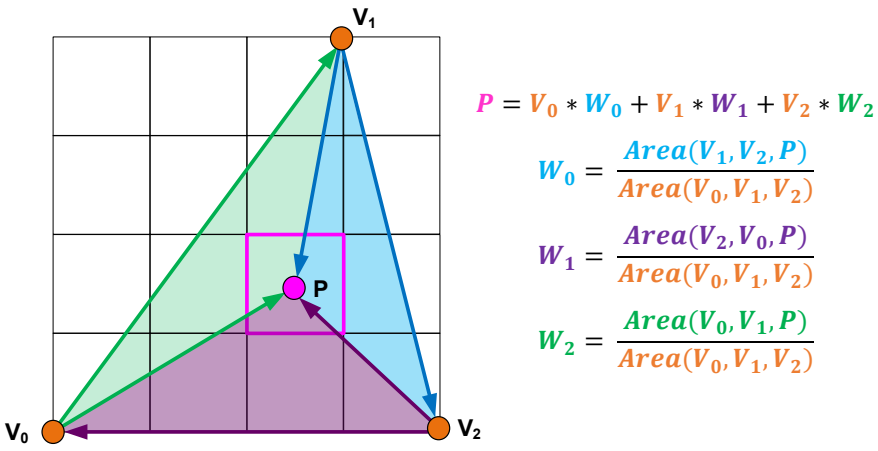


Figure 2.12: Barycentric coordinates computation.

ploying an order-agnostic approach, the Depth Buffer. The Depth Buffer is a memory region which stores the depth of the closest fragment to the camera for every pixel in the frame. Before writing the color of a new fragment into a pixel, the Depth Test is run: its depth is compared with the one stored in the same position and the result is only written if it is closer than the previously visible fragment. As the depth of a fragment corresponds to the interpolated z position attribute, the Depth Buffer is also known as Z Buffer and the Depth Test is also known as Z Test.

In the conceptual model of the Graphics Pipeline the Depth Test is performed after computing the colors of fragments, as such computation may imply modifying a fragment's depth or even discard it. However, most of the time this is not the case, and visibility could have been resolved before color computation without affecting the final result. For this reason, most GPUs employ the *Early* Depth Test functionality to test fragment visibility before proceeding down the pipeline, thus saving time and energy wasted in fragments that will not be visible.

In Tile-Based Rendering, the working set of the Depth Buffer is restricted to only one tile at a time. Therefore, its contents can be stored on an on-chip buffer of as many positions as pixels in the tile and avoid accessing main memory to perform Depth Test operations.

Fragment Processor

The color of the rasterized fragments is computed by executing in the Fragment Processors user-defined shader code that uses the interpolated vertex attributes as inputs. Additional inputs are visible to the programmer and may be employed in the shader, such as constants or the fragment's screen position. Fragment shaders only operate on a single position and do not share temporary information among other fragments. As with the vertex shaders, their execution is independent and can be massively parallelized. The only operation that restricts the execution of fragment shaders is texturing, which requires neighbouring fragments to be bundled into groups of four called *quads*.

Texturing is the process to apply an image to a polygon in order to modify its surface material. Such images are composed of a rectangular grid of pixels called *texels*. Whenever a 3D model with textures is created, a texture coordinate is assigned to each vertex, corresponding to the location of a particular vertex in the image. Texture coordinates are an additional vertex attribute that are

transformed along the other ones in the Geometry Stage and are interpolated for each fragment in the Raster Stage. The transformations in the Geometry Stage might substantially change the sizes of the triangles with respect to the original 3D models, such as when an object is placed far away in the scene and displayed very small. This causes *minification* to occur, where several texels cover a single fragment. Computing the color of a pixel integrating the effect of all texels influencing it cannot effectively be realized in real time, as the number of texels covering a pixel may be very large. Straightforward mechanisms that select a fixed subset of texels may cause aliasing effects whenever a non-representative amount are sampled. The most popular approach to solve minimization is called *mipmapping*, in which textures are stored along with several versions of the original one in progressively lower resolutions. The original texture is called the Level 0 texture and textures in subsequent levels are computed by downsampling the texture in the previous levels, usually just by computing the value of a texel as the average of four neighbour texels. Mipmapping effectively stores pre-computed approximations of the effect that a set of texels has on a pixel, thus reducing aliasing effects caused by minification. Texels are fetched from the level that best approximates a 1-to-1 texel-fragment ratio, known as the *texture level of detail*. The level of detail is computed by checking the difference of the interpolated texture coordinates components across neighbouring fragments in the X and Y axes. Fragments are sent to the processors grouped in 2x2 grids known as *quad* fragments to ease the computation and communication of these gradients.

Texture accesses exhibit both temporal locality (adjacent fragments access adjacent texels) and spatial locality (fragments that are covered by the same texel need to access the same address). Additionally, to further remove aliasing effects, a single fragment tends to fetch several textures (even from different mip levels) and interpolate their texels into a single color value. Consequently, Fragment Processors contain a Texture Cache to exploit locality and reduce the number of accesses to main memory.

Blending

The Blending stage writes the results of the Fragment processors into the Frame Buffer, the region of memory that holds the final colors of the scene and is read by the display. The computed color for a fragment is combined with the previous contents of the Frame Buffer for its position, which allows for transparency effects. This is implemented using the *alpha* channel, an additional value associated to each pixel besides its RGB color and depth. Alpha is a floating point value in the [0,1] range that describes the degree of opacity of a fragment, with 0 representing complete transparency and 1 representing that the fragment is opaque. Alpha blending creates the illusion of transparency by rendering semitransparent fragments on top of the existing scene and attenuating the colors of the fragments behind it. Alpha is used as the attenuating factor, combining the colors in a position by weighting the semitransparent color by α and the previous colors by $1 - \alpha$, as seen in Equation 2.1:

$$Color_{Out} = Color_{New} * \alpha_{New} + Color_{Previous} * (1 - \alpha_{New}) \tag{2.1}$$

In the case that the new fragment is opaque (its alpha is 1), the new color simply replaces the previously stored color. Although the usual way to combine these weighted colors is to add them, the Blending stage can be configured to perform a wide variety of other operations, such as multiplications, subtractions or maximums/minimums.

In Tile-Based Rendering, the working set of the Frame Buffer is restricted to only one tile at a time. Therefore, its contents are stored on the Color Buffer, an on-chip buffer of as many positions as pixels in the tile, which avoids accesses to main memory for Blending operations. Once all the primitives in a tile have been processed, the contents of the Color Buffer are flushed to the Frame Buffer in main memory.

3

Experimental Methodology

This chapter describes the simulation infrastructure used in this thesis to estimate the execution time and energy consumption of graphics applications on a mobile GPU and characterizes the benchmarks used in the experiments.

3.1 Simulation Infrastructure

We employ the Teapot simulation framework [10] to evaluate the proposals described in Chapters 4, 5 and 6. Among the wide variety of available academic GPU simulators, Teapot was selected because of its target to mobile graphics applications, as it offers support for the OpenGL ES API and it models a Tile-Based Rendering architecture in a cycle-accurate manner. Figure 3.1 shows the three layers of the software stack employed by Teapot in order to provide performance and energy consumption statistics.

Application Level

The first step in the Teapot framework logs into a trace all the OpenGL ES commands submitted by a graphics application to the GPU. This step can be performed using two different tools: either Gapid [27] ((1a) in Figure 3.1) or the Android emulator [5] ((1b)). Gapid is an open-source debugging tool for OpenGL ES applications that captures a trace of all API calls made by an application to an Android device such as an smartphone. The Android emulator runs a hardware-accelerated Android virtual device, which allows the execution of unmodified Android applications on a desktop computer. Hardware acceleration enables two elements: on the one hand, the application's command stream is redirected to the host GPU driver, which allows to conveniently intercept it and capture it into a trace file ((2)). On the other hand, applications are run on a ded-

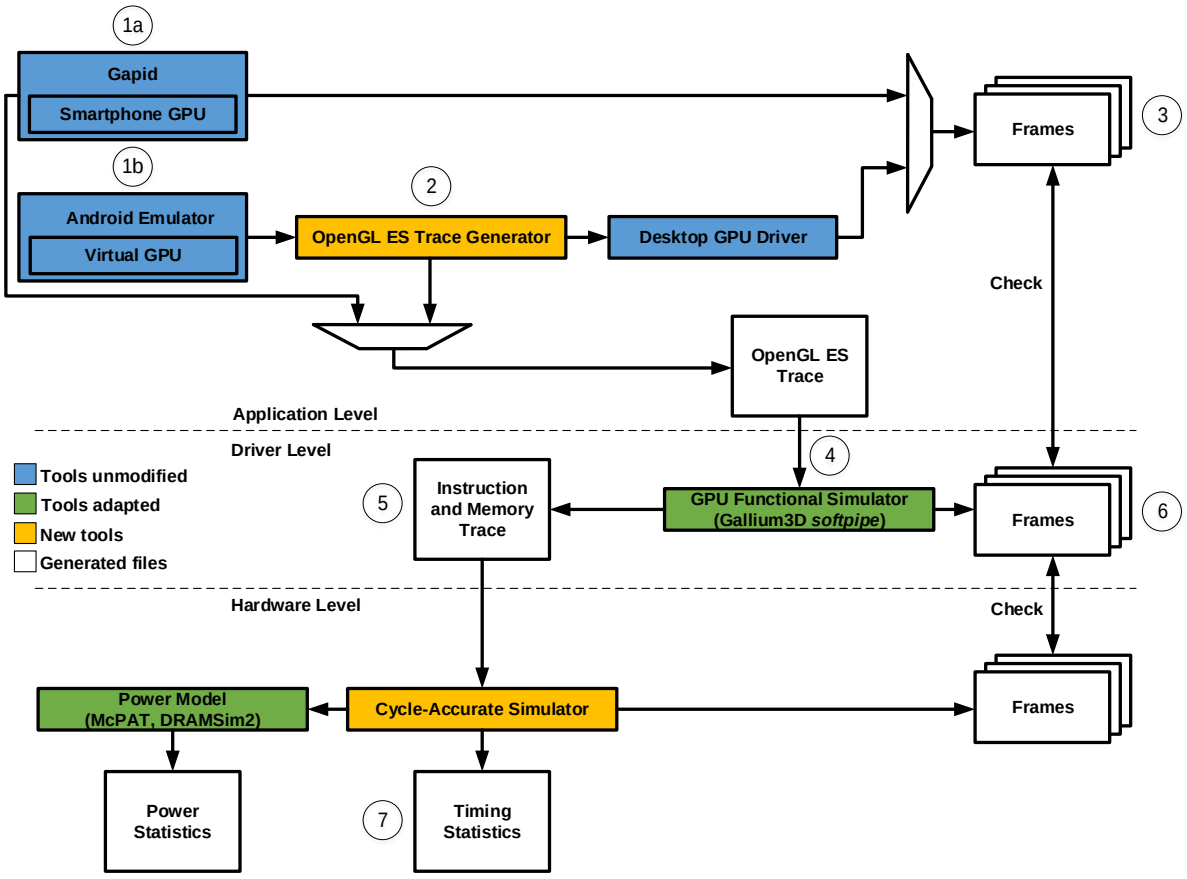


Figure 3.1: Overview of the Teapot simulation infrastructure.

icated graphics card, which eases pressure in the emulation environment and allows the execution of state-of-the-art graphics application to have real-time frame rate and responsiveness. Regardless of the method employed to capture the trace, the GPU renders a set of frames that are saved to check the functionality of the subsequent levels in the framework (③).

Driver Level

The second step in the Teapot framework generates a trace containing all the necessary information to be able to execute a cycle-accurate simulation. The intercepted command trace is fed to Gallium 3D [80] (④), an open-source set of interfaces for developing GPU drivers, configured with the OpenGL ES API as its frontend and a software renderer known as *softpipe* as its backend. The software renderer executes in the CPU all the steps of the Graphics Pipeline that are normally run by the GPU and, therefore, can be instrumented to get a trace of any kind of useful data in any stage of the pipeline, such as transformed vertex attributes, number of fragments rasterized for a primitive, or colors rendered in a position of the framebuffer. In particular, Gallium3D translates the vertex and fragment shaders written in high-level code to an intermediate assembly representation called TGSI [81] and the trace file cointains all executed instructions and memory accesses (⑤).

In addition, the instrumented software renderer generates and stores the frames corresponding to the command trace (⑥). In this way, these images can be compared to the original ones rendered by the GPU to check their correctness.

Hardware Level

The final step in the Teapot framework generates an execution time and energy consumption estimation of the intercepted application by running the trace file in a cycle-accurate simulator and collecting activity factors from all the stages in the pipeline (⑦). Figure 3.2 shows the architecture modelled by the cycle-accurate simulator included in Teapot. It is a Tile-Based GPU closely resembling an ARM Mali-400MP when configured with the parameters listed in Table 3.1.

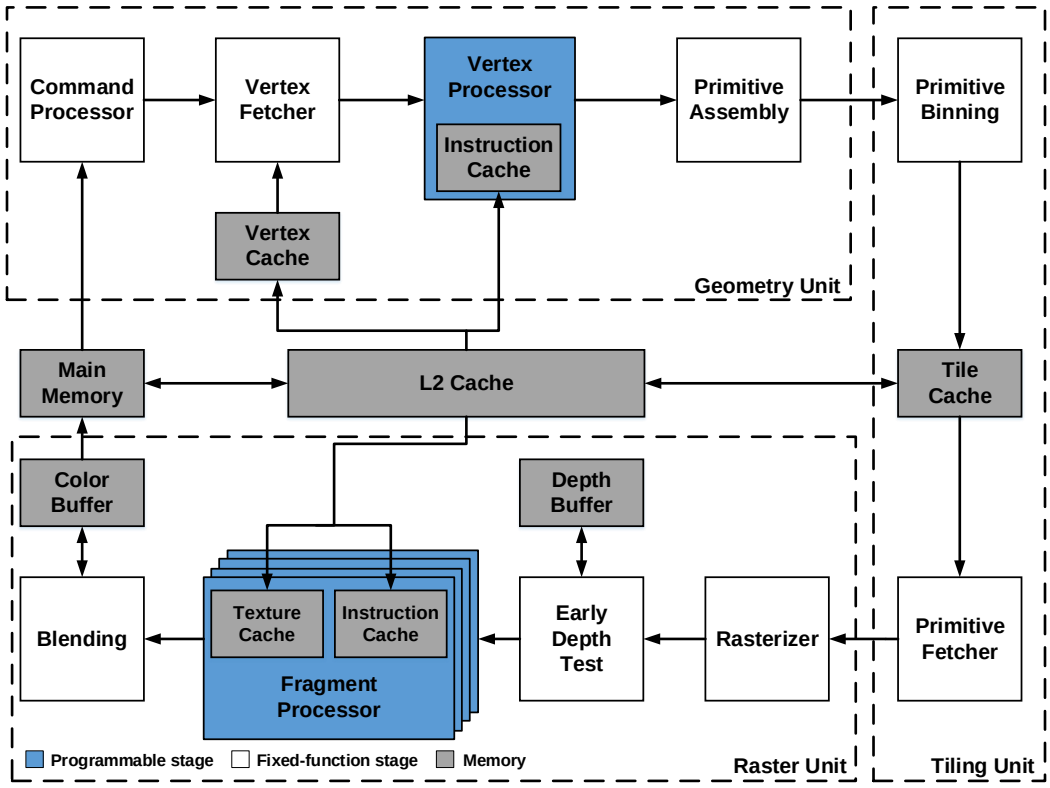


Figure 3.2: Mali-400MP-like architecture modelled by the cycle-accurate simulator.

These parameters are passed to the well-known McPAT power framework [38] when the simulation launches to estimate the static power of every hardware structure (such as queues, processors, caches or registers), as well as the dynamic power of their activations. When the simulation ends, the activity factors of each component are combined with their corresponding individual activation costs to obtain the overall dynamic power of the GPU. Then, the reported execution time is used to compute the energy consumption.

Teapot employs the widely-used DRAMSim2 [63] to model timing and energy of the main memory system. DRAMSim2 cycle-accurately models DRAM ranks, banks, memory channels and the memory controller of DDR2 and DDR3 variants, and uses the model described by Micron [48]

to compute the power consumption of each bank.

Table 3.1: GPU Simulation Parameters.

Baseline GPU Parameters	
Tech Specs	400 MHz, 1 V, 32 nm
Screen Resolution	1196x768 (Chapters 4 and 5) 1920x1080 (Chapter 6)
Tile Size	16x16 pixels
Main Memory	
Latency	50-100 cycles
Bandwidth	4 B/cycle (dual channel LPDDR4)
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
On-Chip Buffers	
Color Buffer	256 entries, 32 bits/entry
Depth Buffer	256 entries, 24 bits/entry
Non-programmable stages	
Primitive Assembly	1 triangle/cycle
Rasterizer	16 attributes/cycle
Early Z test	32 in-flight quad-fragments
Programmable stages	
Vertex Processor	1 vertex processor
Fragment Processor	4 fragment processors

3.1.1 Improvements to the baseline infrastructure

During the development of this thesis, the three layers of the Teapot infrastructure have been adapted (listed chronologically):

1. **Hardware Level:** DRAMSim2 has been slightly extended to model the timing and energy consumption of LPDDR4, the most extended SDRAM variant targeted at mobile devices.

2. **Driver Level:** The techniques described in Sections 4 and 5 require the cycle-accurate simulator to use intermediate data resulting of the Geometry Pipeline. The original infrastructure did not generate it because most of the cycle-accurate simulation does not reproduce functional behavior. The software renderer has thus been instrumented to generate such data (transformed vertex attributes and draw call uniforms) and include it in the final trace file.

3. **Application Level:** Gapid was launched in 2018 by Google and the Teapot infrastructure was adapted to use it as its command trace interceptor. This change improved the ease to acquire new benchmarks, as many tasks in the original interceptor (such as writing wrappers for newly seen commands or splitting the stream into frames) had to be manually performed for each benchmark to consider. Additionally, applications can be intercepted directly from a smartphone and not the Android emulator, which increases the variety of input mechanisms games can have (e.g., motion controls) and the resulting frame rate. The benchmarks used in the technique described in Section 6 are intercepted using Gapid instead of the Android Emulator.

3.2 Benchmark Set

The proposals in this thesis have been evaluated using a set of commercial Android applications. While it is common to rank the performance of different GPUs by using the results that synthetic benchmarks such as GFXBench provide, these kind of benchmarks test capabilities in a manner that substantially deviates from the real use that mobile graphics applications have. They test individual parts of the system by stressing them to see their theoretical peak performance without taking a holistic approach (e.g. without regarding important contextual components such as the operating system or the battery life), which do not represent the whole system's behavior. In particular, graphics benchmarks tend to compute the number of frames a GPU can render in a given time period. Not only the result is generally uninteresting (being able to render 300 frames instead of 200 does not affect user experience), but also the scenes present an unrealistic number of small triangles in order to increase the workload, whereas the geometry-to-fragment ratio is much more skewed towards the latter in daily-used applications. Therefore, it was decided to deviate from synthetic benchmarks and instead use applications that are representative of the current mobile gaming landscape. The selection criteria is based on two points: on the one hand, applications on the set must have a large number of downloads in the App Stores, as popular games are more likely to be played by the average user. On the other hand, applications in the set must be diverse in terms of workload complexity (games that stress the GPU exist, but most of the games that are played have rather simple graphics), graphics type (2D or 3D) and genre (from basic puzzle games with still scenes to fast paced first person shooters) so that the presented proposals are validated in a wide variety of scenarios.

Table 3.2 lists the benchmarks used throughout the experimental evaluations of this work. For each application, 100 frames of archetypal execution have been captured (e.g. no loading or pause screens) at a consistent frame rate of 20 to 30 frames per second. The benchmark suite is different for Chapter 6 than for Chapters 4 and 5, since the infrastructure update allowed the use of more contemporary applications. The table gives a summary of why these benchmarks have been chosen: popularity and variety. It can be seen that most games have been downloaded tens of millions of

times, with some applications being the most famous mobile games of all time, even surpassing the half billion download mark. Furthermore, the suite is noticeably diverse in terms of gaming genres and type of graphics.

The applications have been selected not only because of their genre variety, but also for they workload variety. This can be seen in Tables 3.3 and 3.4 which show, respectively, some characterization of the Geometry and Raster Pipelines. On the Geometry side, Table 3.3 lists the averages of: number of drawcalls per frame (the number of times the Application sends a collection of vertices with an associated state to the GPU to render), number of vertices per drawcall, number of attributes per vertex, number of assembly instructions executed by vertex shaders (static, as in these benchmarks there are no loops), number of primitives per drawcall and number of assembled primitives binned per tile.

Used in	Benchmark	Genre	Type	Downloads in Google Play (Millions)
Chapters 4 and 5	300	Hack and Slash	3D	50
	Air Attack	Arcade	3D	1
	Angry Birds	Puzzle	2D	100
	Army Men Strike	Real-Time Strategy	2D	10
	Avenger Legends	Role-Playing Game	2D	1
	Candy Crush Saga	Match-Three Puzzle	2D	500
	Castle Defense	Tower Defense	2D	10
	Clash of Clans	MMO Strategy	2D	500
	Crazy Snowboard	Arcade	3D	5
	Cut the Rope	Puzzle	2D	100
	Dude Perfect	Puzzle	2D	10
	Hay Day	Simulation	2D	100
	Hopeless: The Dark Cave	Action Survival	2D	5
	Magic Touch: Wizard for Hire	Arcade	2D	5
	Modern Strike	First-Person Shooter	3D	50
	RedSun	Real-Time Strategy	2D	1
	Temple Run	Endless Runner	3D	100
	Tigerball	Puzzle	3D	10
	Where's my water	Puzzle	2D	100
	World of goo	Physics Puzzle	2D	1
Chapter 6	Brawl Stars	Beat'em Up	3D	100
	Clash Royale	Real-Time Strategy	2D	100
	Dragon Ball Z: Dokkan Battle	Role-Playing Game	2D	10
	Guns of Boom	First-Person Shooter	3D	50
	Hearthstone	Collectible Card Game	2D	10
	Merge Dragons	Puzzle	2D	10
	Minecraft	Sandbox	3D	50
	Rise of Kingdoms: Lost Crusade	Real-Time Strategy	2D	10
	Sonic Dash	Endless Runner	3D	100
	Toy Story Drop	Match-Three Puzzle	2D	1

Table 3.2: Benchmarks set.

The benchmark suite covers a wide range of geometric complexity, with games such as *Castle Defense* only processing a few hundred vertices per frame while *Minecraft* fetches and transforms

Benchmark	Drawcalls per Frame	Vertices per Drawcall	Attributes per Vertex	Vertex Shader Instructions per Vertex	Primitives per Drawcall	Primitives per Tile
300	133.47	336.08	3.30	25.57	299.98	19.80
Air Attack	21.10	139.87	2.67	7.60	275.97	4.50
Angry Birds	14.00	24.31	2.05	6.02	8.06	2.30
Armymen	78.53	20.09	2.99	7.99	10.05	4.00
Avenger Legends	132.43	33.97	3.00	8.00	32.52	5.70
Candy Crush Saga	13.45	99.20	2.99	12.83	49.60	3.30
Castle Defense	80.51	5.65	3.00	8.00	2.88	1.80
Clash of Clans	87.49	46.24	3.09	6.40	61.99	6.70
Crazy Snowboard	22.68	133.58	2.94	13.48	160.68	3.30
Cut the Rope	91.35	8.14	1.99	5.99	4.89	2.60
Dude Perfect	116.84	13.31	3.00	8.00	7.16	6.70
Hayday	28.75	64.85	2.52	5.15	88.73	3.10
Hopeless	84.59	106.11	2.95	3.95	35.37	5.40
Magic Touch	26.86	36.39	2.99	8.34	18.22	3.20
Modern Strike	24.31	422.20	2.13	8.58	352.75	4.80
Redsun	6.12	320.28	2.00	12.98	159.81	2.60
Temple Run	20.39	410.25	2.76	31.16	757.04	4.20
Tigerball	45.03	122.63	2.98	162.11	140.42	4.90
Where's my Water	92.06	93.65	3.45	7.45	91.65	13.50
World of Goo	37.45	28.04	2.99	6.99	14.02	4.30
Brawl Stars	178.40	643.67	6.85	26.13	372.08	9.28
Clash Royale	71.40	48.95	2.65	5.52	65.02	6.47
Dragon Ball: Dokkan Battle	229.14	13.22	2.88	7.95	6.54	16.24
Guns of Boom	41.12	1160.24	2.94	29.53	996.89	8.4
Hearthstone	141.98	138.56	3.10	37.04	166.68	14.7
Merge Dragons	37.58	111.79	3.01	13.52	64.98	5.4
Minecraft	500.16	936.74	3.95	22.19	504.18	21.5
Rise of Kingdoms	61.83	247.11	3.01	33.18	106.42	3.3
Sonic Dash	87.84	1019.16	3.68	80.98	850.72	14.5
Toy Story Drop	45.20	125.55	4.48	58.87	124.89	5.5

Table 3.3: Characterization of the geometry workload processed by the considered benchmarks.

hundreds of thousands. The same behavior can be observed in the number of primitives assembled and processed in a frame. Vertices are sent to the GPU in a myriad of ways: *Redsun* processes very few drawcalls containing hundreds of vertices, while *Dragon Ball* parses hundreds of drawcalls containing around a dozen vertices each. Games like *Angry Birds* receive very few amounts of drawcalls and vertices, while *300* processes a large amount of both drawcalls and vertices. In comparison with general purpose programs, the number of shader program instructions is very small, with most benchmarks executing less than 20 instructions. A significant difference in shader complexity can be observed between 2D and 3D games, with the shaders of most 2D games such as *Castle Defense* or *Armymen* being composed by less than 10 instructions while the number of instructions executed by the shaders of 3D games such as *Sonic Dash* or *Tigerball* is one or two magnitudes larger. The overhead of tiling is modest in most games, requiring only the storage and later fetch of only around 5 primitives per tile. However, in benchmarks such as *300* or *Merge Dragons* that contain lots of small primitives, that number is increased 4 times, significantly increasing the communication with main memory.

Benchmark	Fragments per Primitive	Fragment Shader Instructions per Fragment	ALU to TEX Ratio	Texels per Fragment	Overshading
300	506.27	4.03	7.34	4.01	6.4
Air Attack	1580.97	3.26	4.00	7.00	2.6
Angry Birds	21999.40	4.09	5.84	3.03	3.1
Armymen	5409.53	4.83	5.15	4.13	3.2
Avenger Legends	663.46	4.13	5.92	3.14	3.1
Candy Crush Saga	2737.04	4.00	4.03	4.38	1.9
Castle Defense	5590.10	4.00	4.00	4.33	1.4
Clash of Clans	1158.47	3.45	3.45	4.43	3.9
Crazy Snowboard	2461.45	2.88	7.64	3.06	2.2
Cut the Rope	3756.69	3.99	4.01	4.32	1.8
Dude Perfect	5691.76	4.00	4.00	3.61	4.5
Hayday	1387.63	3.47	3.47	4.49	6.2
Hopeless	487.78	5.61	5.61	4.78	1.6
Magic Touch	5949.46	3.81	6.93	2.70	2.8
Modern Strike	994.08	3.41	5.40	3.89	5.2
Redsun	4236.86	5.83	4.57	5.61	2.8
Temple Run	1582.31	5.85	10.04	5.76	3.6
Tigerball	504.96	7.91	8.55	4.48	2.8
Where's my Water	1202.97	4.02	4.15	4.75	7.5
World of Goo	7134.39	2.98	4.00	4.33	2.4
Brawl Stars	117.28	10.96	6.29	13.61	1.6
Clash Royale	12284.43	6.93	4.03	7.43	4.2
Dragon Ball: Dokkan Battle	23331.35	4.11	3.21	2.35	9.1
Guns of Boom	1157.35	1.79	1.16	3.73	4.1
Hearthstone	596.10	3.56	4.67	6.19	3.8
Merge Dragons	3295.82	7.62	5.87	5.66	3.4
Minecraft	156.07	3.04	5.91	4.93	1.6
Rise of Kingdoms	3108.12	5.09	10.57	4.66	2.4
Sonic Dash	870.54	11.40	1.89	6.39	4.2
Toy Story Drop	1515.73	5.33	5.47	4.34	3.5

Table 3.4: Characterization of the fragment workload processed by the considered benchmarks.

Regarding the Raster Pipeline, Table 3.4 lists the averages of: number of generated fragments per primitive, number of static assembly instructions executed by fragment shaders, number of ALU instructions per texture instructions in such programs, number of fetched *texels* (texture elements) in a texture instruction and overshading (number of times a position in the Color Buffer is overwritten by a newly processed fragment of the same frame).

It can be seen that the number of fragments is generally three orders of magnitude larger than the number of vertices, which makes the Raster Pipeline a very significant contributor in time and energy consumption for these benchmarks. The fragment shader programs contain even less instructions than the vertex shader counterparts, with only a few 3D games such as *Brawl Stars* or *Sonic Dash* executing more than 10. In this benchmark suite, fragment programs only access memory through texture operations and thus, the column showing the ratio between ALU and Texture instructions indicates how memory-intensive the programs are. Most of the shaders

executed by these benchmarks have a high ratio (in fact, only contain a single texture instruction), allowing the processors to make progress for other warps while serving memory requests. Each of those texture instructions fetches more than one texel, as reconstruction filters are employed to obtain a color for a fragment not perfectly-aligned with the texture maps. The most common used methods [31] are nearest-neighbor, bilinear filtering, trilinear filtering and anisotropic filtering which require, respectively, 1, 4, 8 and 16 texels to be fetched per fragment. Different filters are used across the benchmarks, from heavy uses of nearest-neighbor in *Angry Birds* and *Armymen* to combinations of trilinear and anisotropic in *World of Goo*. Finally, Table 3.4 shows that most benchmarks have an overshading factor much larger than 1, revealing that a significant amount of energy and time is wasted on processing redundant fragments for the same position in the screen.

4

Rendering Elimination

The GPU and, in particular, Fragment Processing, is the biggest contributor to energy consumption in real time rendering, as many memory accesses and computations need to be performed to obtain a color for every pixel in the screen [3, 12, 54]. Due to frame coherence, many tiles produce the same colors across consecutive frames, and current GPUs waste energy by computing them. Figure 4.1 shows this phenomenon by plotting the average percentage of equal tiles between two consecutive frames for a set of commercial Android games.

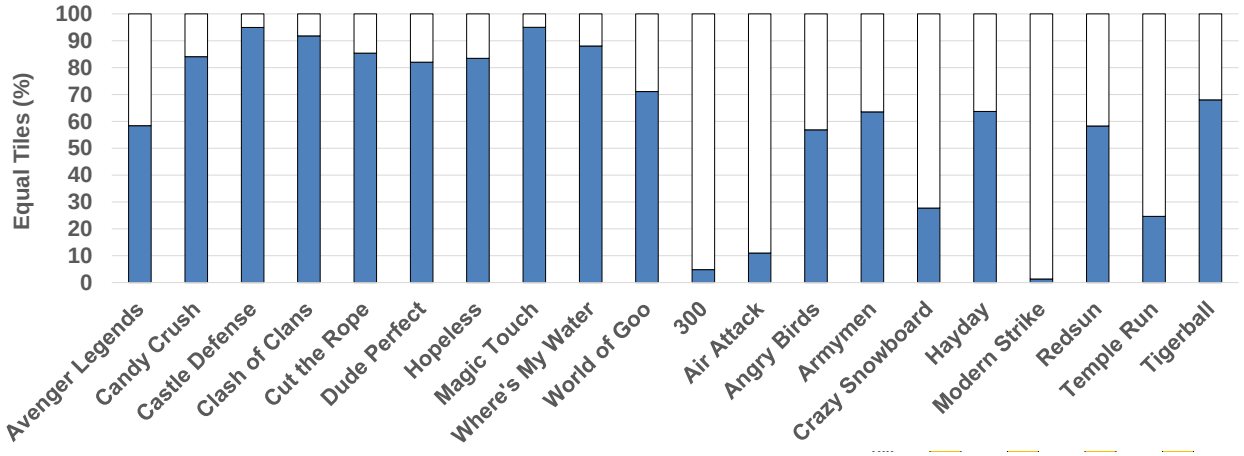


Figure 4.1: Percentage of tiles producing the same result (the color is equal for all their pixels) as the preceding frame across 50 consecutive frames.

In games with moderate camera movements (*Avenger Legends* to *World of Goo*), it is common for more than 80% of tiles to produce the same color as in the preceding frame. This feature can also be found, albeit less frequently, in games where the scene is in continuous motion (*300* to

Tigerball). This chapter presents Rendering Elimination, a technique that eliminates the processing of such redundant tiles, and describes it at a micro-architectural level. Rendering Elimination is applied on top of a Tile-Based Rendering GPU and shown to reduce energy consumption by 37% and execution time by 33%, widely improving state-of-the art approaches to reduce redundant computations.

4.1 Early Discard of Redundant Tiles

4.1.1 Rendering Elimination Overview

The Raster Pipeline takes as inputs the scene constants and the attributes of all the primitives that overlap a tile, and produces a color for each pixel belonging to that tile. The execution of the Raster Pipeline does not generate any side effects, i.e., it does not produce any changes in state variables. Such changes are performed by API calls that can be easily tracked by the driver. Consequently, redundancy for a tile can be determined in advanced by comparing its inputs for the current frame against the inputs for the previous frame: if the two input sets match, the outputs will also be equal.

Because of the large volume of these sets, storing them in main memory would be extremely inefficient, even with the support of a cache, because the reuse distance between them is an entire frame. Instead, a more efficient approach based on computing a signature for the inputs of the tile and storing it in a local buffer is used. This buffer, called *Signature Buffer*, contains the signatures of all the tiles of the previous frame and the current one. Figure 4.2 depicts the Graphics Pipeline flow with the added Signature Buffer.

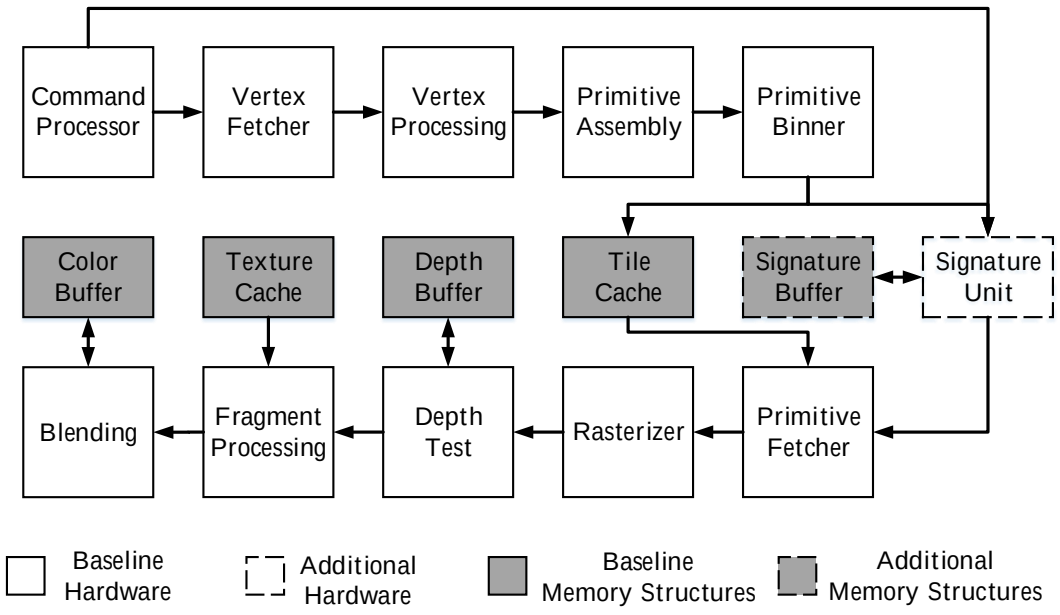


Figure 4.2: Graphics Pipeline including Rendering Elimination.

The Signature Unit computes the signatures employing the primitives that the Primitive Binner produces and inserts them into the Signature Buffer. At the same time, the Primitive Binner fills the Parameter Buffer with the data of such primitives, including identifiers of the tiles that contain them. After the geometry of the frame has been processed, the Signature Buffer holds signatures for the inputs of all the tiles. Hereafter, whenever a tile is scheduled in the Raster Pipeline, its Signature Buffer entry is checked: if the current frame signature matches that of the previous frame, the Raster Pipeline execution is skipped and the Frame Buffer locations for that tile are not updated. Otherwise, the Raster Pipeline is executed normally.

4.1.2 Implementation Requirements

The signature of a tile is computed by hashing a list of all the inputs of a tile: this includes the vertex attributes and scene constants associated to all the primitives that overlap the tile. Such inputs are produced either by the Command Processor when setting scene constants for a drawcall or by the Primitive Binner when sorting primitives and storing their vertex attributes into the Parameter Buffer. The stream of primitives produced by the Geometry Pipeline, however, is generated in the order that the GPU received the drawcalls, which is generally not the order in which they appear in the screen. In fact, any primitive from the stream could overlap any number of tiles. This causes that the complete list of inputs for a tile is not known until all the geometry of the scene has been processed. A straightforward implementation that starts computing the signatures when the Geometry Pipeline has processed the whole frame would not be practical. Since vertex attributes are stored in the Parameter Buffer (residing in off-chip memory), retrieving them in order to compute a signature for the tile would require significant time and energy overheads and delaying the execution of the Raster Pipeline.

To be effective, Rendering Elimination computes the signatures for the current frame in an *incremental* approach. Whenever a primitive is sorted, the temporary signatures for each tile that it overlaps are read. The new signature for each tile is constructed by combining the temporary signature with either the scene constants or the attributes of the vertices of the current primitive and, afterwards, it is rewritten in the appropriate Signature Buffer entry. This on-the-fly signature computation is overlapped with other Geometry Pipeline stages, resulting in minimal overheads in execution time, as shown in Section 4.3.

The signature function employed by Rendering Elimination is CRC32 [56]. While a plethora of other mechanisms exist, CRC32 outperforms well-known hashing approaches such as XOR-based schemes, as will be shown in Section 4.3. When using CRC32, not a single instance of hashing collisions have been observed. Moreover, as a widely-used error detection code, CRC has been extensively researched in the literature and efficient techniques have been developed [70] that allow for an incremental and parallel CRC computation based on Look-up Tables, as outlined below.

4.1.3 Incremental CRC Computation

As proven in [70], the CRC of a message can be computed even if its length is not known a priori by breaking it down into several submessages and computing the CRC of those submessages

independently. Given a message A, composed by concatenating submessages $A_1...An$, of lenghts $b_1...b_n$ bits, the CRC of A can be computed as:

Algorithm 1 Incremental CRC Computation

$CRC_A = 0$
for submessage A_i in A **do**
  $b = length(A_i)$
  $CRC_{A_i} = ComputeCRC(A_i)$
  $CRC_{Temporary} = ComputeCRC(CRC_A << b)$
  $CRC_A = CRC_{A_i} \oplus CRC_{Temporary}$
end for

That is, the CRC of the first submessage A_1 is computed. When the length b of the following submessage A_2 is known, the CRC of the two submessages (a bit string formed by concatenating A_1 and A_2) is computed by first computing the CRC of A_2, left-shifting the CRC of A_1 by b bits, computing the CRC of this shifted message, and combining both CRCs via an XOR function. By means of this procedure, CRCs of partial messages of increasing length are computed: first, the CRC of A_1, then the CRC of the concatenation of A_1 and A_2, then the CRC of the concatenation of A_1, A_2 and A_3, and so on, until the last submessage A_n is reached and, therefore, the CRC of the concatenation of the submessages corresponds to the CRC of the original message.

4.1.4 Table-Based CRC Computation

Each iteration in Algorithm 1 would require several cycles if the CRC computation was implemented using the basic Shift Register mechanism [45]. A faster alternative is to use a Look-up Table (LUT) loaded with precomputed CRC values for all possible inputs. However, this approach is unfeasible in terms of storage requirements, since a message of length n requires a LUT of 2^n entries. As shown in [70], a message B of n bits, being n multiple of 8, can be broken into k 1-byte blocks $B_1...B_k$ ($n = 8 \times k$) and use a small LUT to efficiently compute the CRC of each block.

Each LUT takes as input a block B_i and computes the CRC of a message corresponding to left-shifting B_i by $k - i$ bytes. Namely, the first LUT computes the CRC of a message consisting of block B_1 followed by $k - 1$ bytes of zeros, the second LUT computes the CRC of a message consisting of block B_2 followed by $k - 2$ bytes of zeros and the k^{th} LUT computes the CRC of a message consisting of block B_k. The results of the k LUTs are combined into one CRC via an XOR function.

Since each LUT has 2^8 entries and each entry contains a precomputed CRC value, the size of each LUT is 1 KB and, consequently, computing the CRC of a message of length n bits has a storage cost of k KB.

4.1.5 Tile Inputs Bitstream Architecture

Rendering Elimination determines if the colors of two tiles are going to be the same by comparing the signature of their inputs. The inputs of a tile are the vertex attributes of the primitives

that overlap it and the set of scene constants associated to those primitives. In order to render primitives, the GPU receives a series of commands that define the state of the pipeline (shaders, textures, constants) and drawcalls, which contain a stream of vertices to be processed with the defined state.

Each drawcall can generate any number of primitives and each primitive can overlap any number of tiles. Therefore, the input of a tile consists of a sequence of blocks, one for every drawcall that contains the primitives that overlap this tile. Each block is, in turn, composed of several subblocks: a first subblock corresponding to the constants defined in the drawcall followed by a list of subblocks that correspond to the attributes of the primitives that overlap this tile. Since both the number of primitives overlapping a tile and the number of attributes of those primitives is not fixed, neither are the lengths of the blocks nor is the length of the subblocks.

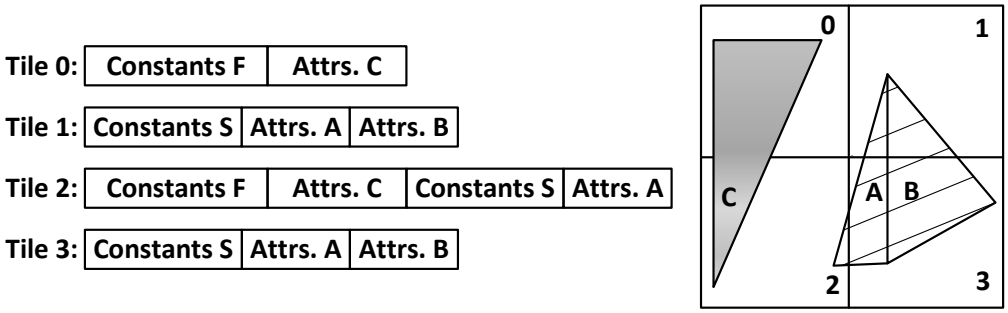


Figure 4.3: Example of input message.

Figure 4.3 provides an example of the described tile inputs for four tiles and the primitives of two drawcalls: Drawcall F (fill) and Drawcall S (stripes). Drawcall F generates Primitive C, which overlaps Tiles 0 and 2. Therefore, the inputs of Tiles 0 and 2 contain the block of Drawcall F, composed of a set of constants and the attributes of Primitive C. Drawcall S generates two primitives, Primitives A and B. These two primitives overlap Tiles 1 and 3, so the inputs of Tiles 1 and 3 contain the block of Drawcall S, composed of a set of constants and the attributes of both primitives. Note that, while two primitives of Drawcall S overlap Tiles 1 and 3, the set of constants of the drawcall is only considered once for those tiles. Primitive A also overlaps Tile 2, so the set of constants of Drawcall S as well as the attributes of Primitive A are added to Tile 2's inputs.

Besides scene constants, primitives have other global associated data that affects the color of a fragment: the shader program and the textures to be used within. Rendering Elimination does not include these in the tile signature, since changes to such global data are not common. In our benchmarks, we have observed that shaders and textures remain constant for thousands of frames. Moreover, loading new shaders and textures is done through API calls (such as *glShaderSource* and *glTexImage2D*, for instance) and, therefore, are registered by the driver. Whenever such infrequent API calls occur, Rendering Elimination is disabled for the current frame. Besides this, Rendering Elimination could also be disabled during one frame periodically to guarantee Frame Buffer refreshing. Rendering Elimination should also be temporarily disabled by the driver for scenes that use multiple render targets, as it is specifically targeted to an important segment of less sophisticated applications that cover a large fraction of the mobile market.

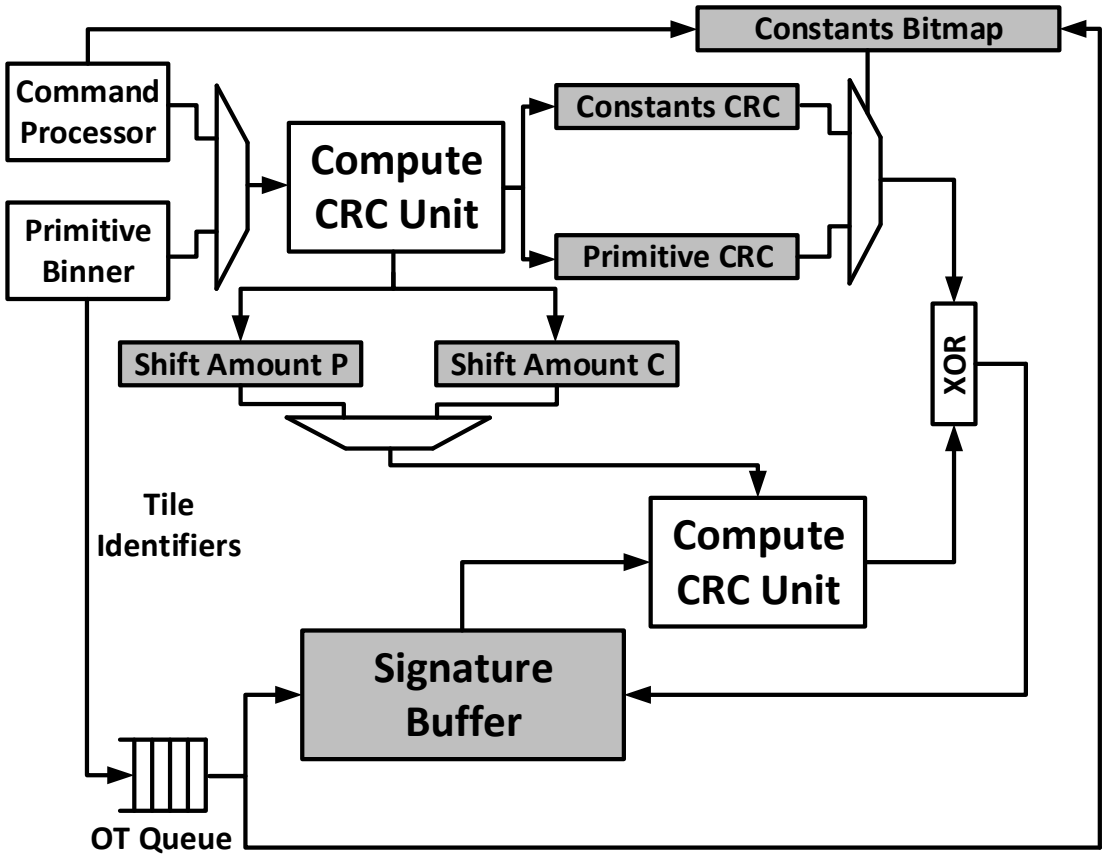


Figure 4.4: Signature Unit block diagram.

4.2 Implementation

4.2.1 Signature Unit Architecture

The message that has to be signed for a tile consists of a sequence of blocks, containing either scene constant data or vertex attribute data. The number of blocks of a message is not known until all the geometry of the frame is processed and, therefore, Rendering Elimination uses the incremental signature computation described in Algorithm 1.

The Signature Unit (SU), the piece of logic responsible for the incremental computation of the CRCs of tiles, is shown in Figure 4.4. Whenever the SU receives a new data block, it computes its CRC and updates the CRC of all the tiles overlapped by the primitive associated to that block.

Let us consider first the case of vertex attributes, which are blocks sent to the SU by the Primitive Binner. The SU computes the signature of all the vertex attributes of a primitive using the *Compute CRC* unit, and the resulting CRC32 (CRC_{A_i} as described by Algorithm 1) is stored in the *Primitive CRC* register. Since the number of attributes in a primitive is variable, the Compute CRC unit stores the length of the signed block (b in Algorithm 1) in the *Shift Amount P* register.

While the SU computes the CRC of a primitive, the Primitive Binner inserts into the *OT Queue* a list of identifiers of the tiles overlapped by the primitive.

After computing the signature of a primitive, the SU traverses the list of overlapped tiles and updates each tile signature by combining it with the primitive signature. It pops in sequence each entry from the head of the OT Queue and uses this tile *id* to read the corresponding CRC from the Signature Buffer, which is then sent to the *Accumulate CRC* unit. This unit receives as inputs the previous CRC for a tile and the length of the primitive message signed by the Compute Unit. The Accumulate CRC unit computes the CRC of the message that results by left-shifting the previous CRC as many bits as the received length. This CRC corresponds to $CRC_{Temporary}$ in Algorithm 1. Finally, the results of the Compute and Accumulate units are bitwise xored to obtain the new CRC for the tile (CRC_A in Algorithm 1) and the new signature is written back to the Signature Buffer.

The SU can also receive data blocks from the Command Processor, which correspond to scene constants. The signature computation of the constants of a drawcall is done in the same form as the signature computation of the vertex attributes of a primitive: the Compute Unit generates a CRC32 and the length of the signed message and stores them in two registers: *Constants CRC* and *Shift Amount C*, respectively. In order to combine the signature of the constants with the signature of the attributes, several issues need to be addressed. First, every drawcall may define its own set of constants which only affect to that drawcall. Consequently, the Constant CRC register only has to be combined with the CRC of the tiles affected by that drawcall. Besides this, even though multiple primitives of the same drawcall may overlap the same tile, the Constant CRC should be considered only once per tile.

Rendering Elimination uses a bitmap to solve these issues. The bitmap has a length equal to the number of tiles that the Frame Buffer is divided into. If a position of the bitmap is set, it means that the Constant CRC has already been combined into the signature for that tile. Whenever the GPU receives a new set of constants after having processed one or more drawcalls, the bitmap is cleared and the constants are signed and stored in the Constant CRC register. For all the following primitives, for every tile identifier popped from the OT Queue, the bitmap is queried to check whether that tile has already combined the signature of the constants into its signature. If so, the previous CRC of the tile is only updated with the value stored in the Primitive CRC register. Otherwise, the bit in the bitmap position corresponding to that tile is set and the previous CRC of the tile is updated twice: first with the contents of the Constants CRC, and second with the Primitive CRC, by making the Accumulate CRC unit to select the appropriate shift amount in each step.

All the structures in the Signature Unit have been modelled using McPAT components in order to obtain an area and power estimation: the parameters of the SRAMs present in the framework can be set to accurately describe the Signature Buffer, the Overlapped Tiles Queue, the constant bitmap as well as the table-based CRC computation of the Compute CRC and Accumulate CRC Units, which is explained in detail in the following section. McPAT also is able to model simpler components such as logic gates and D flip-flops, which are used to account for the power and area of the XOR gates and the registers present in the Signature Unit.

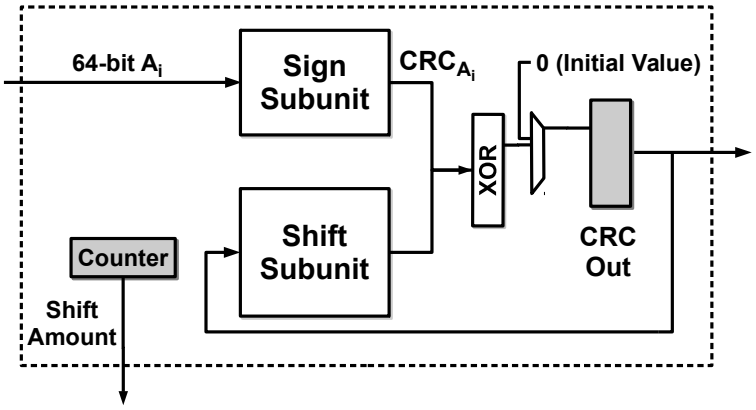


Figure 4.5: Compute CRC Unit block diagram.

4.2.2 Compute CRC Unit and Accumulate CRC Unit

Algorithm 2 Compute CRC Unit, Incremental Computation

$CRC_{Out} = 0$
$ShiftAmount = 0$
for 64-bit subblock A_i in submessage A **do**
  $CRC_{A_i} = ComputeCRC(A_i)$
  $CRC_{Temporary} = ComputeCRC(CRC_{Out} << 64)$
  $CRC_{Out} = CRC_{A_i} \oplus CRC_{Temporary}$
  $ShiftAmount = ShiftAmount + 1$
end for

The **Compute CRC unit** implements the first two steps in the loop of Algorithm 1, computing the CRC of a block consisting of a primitive or a set of constants and determining the length of the block. Since the length of such blocks is not fixed, the Compute CRC unit is architected to incrementally compute the CRC32 of a block by breaking it into subblocks of fixed length (64 bits) and recursively applying Algorithm 1. The resulting procedure is detailed in Algorithm 2. Namely, the Compute CRC unit has a similar internal structure as the SU, as shown in Figure 4.5. It consists of two subunits and the CRC_{Out} register (initialized to zero). The *Sign* subunit computes the CRC32 of a fixed-length subblock and stores it into the CRC_{Out} register after a bitwise XOR with the result of the *Shift* subunit. In parallel, the Shift subunit computes the CRC32 of the message resulting by left-shifting 64 bits the contents of the CRC_{Out} register. This process is repeated for each 64-bit subblock in the input data block received by the Compute CRC unit. The control logic of the Compute CRC unit counts the number of signed subblocks and communicates it to the Accumulate CRC unit using registers Shift Amount P (for Primitives) and Shift Amount C (for Constants), shown in Figure 4.4.

The **Accumulate CRC unit** implements the third step in the loop of Algorithm 1, that computes the CRC of a message consisting of the partial CRC of a tile (stored in the Signature Buffer) left-shifted by as many zeros as the length of the block to accumulate (the one fed to the Compute CRC unit). Since the length of this block is variable, it is also variable the amount to shift,

Algorithm 3 Accumulate CRC Unit, Incremental Computation

$CRC_{Accum} = SignatureBuffer[tile]$
for $k \leftarrow 1$ **to** $ShiftAmount$ **do**
  $CRC_{Accum} = ComputeCRC(CRC_{Accum} << 64)$
end for

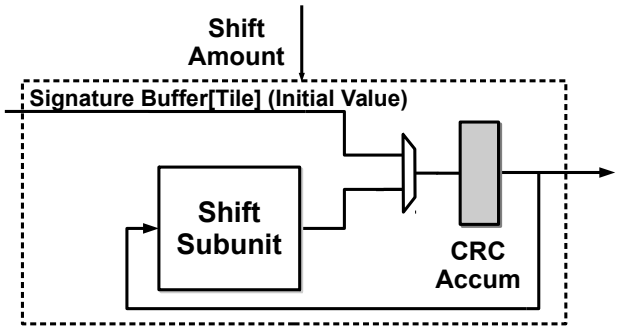


Figure 4.6: Accumulate CRC Unit block diagram.

hence the length of the resultant message to be signed by the Accumulate CRC unit. Therefore, this unit follows an incremental procedure to compute the CRC, as detailed in Algorithm 3. Note that, while the Accumulate CRC unit follows the same incremental approach as the Compute CRC unit, the accumulated blocks are always zero (they come from a left shift). Therefore, each iteration only requires to shift and re-sign the CRC32 computed on the preceding iteration and, consequently, the Accumulate CRC unit only consists of a Shift subunit, as shown in Figure 4.6.

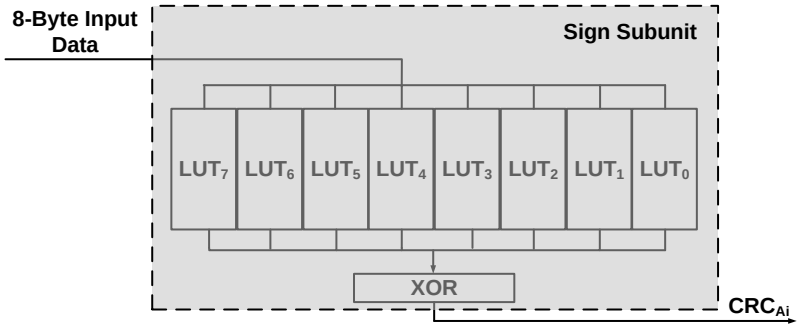


Figure 4.7: Architecture of the Sign subunit.

Figure 4.7 shows the **Sign subunit** architecture, which computes the CRC32 of a 64-bit subblock using the table-based approach described in Section 4.1.4. Each byte in the subblock is independently processed by accessing a specific LUT. The output of the Shift subunit is the bitwise XOR of the results of the 8 LUTs.

Figure 4.8 shows the **Shift subunit** architecture, which computes the CRC32 of the 64-bit message that results from a 32-bit input block shifted with 32 zeros. The design is analogous to the Sign subunit, and uses the table-based approach described in Section 4.1.4.

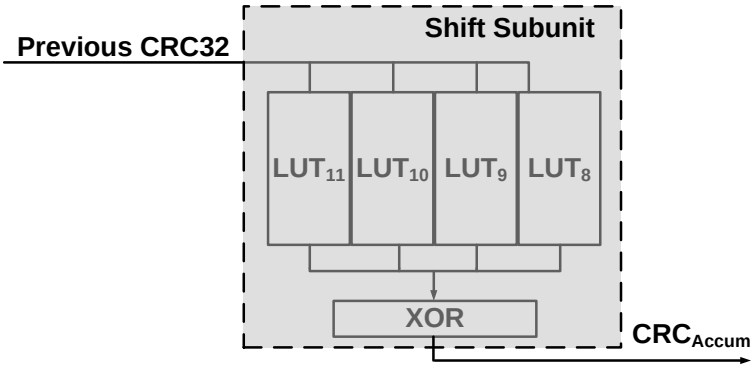


Figure 4.8: Architecture of the Shift subunit.

The choice of the subblock size for the Compute CRC unit is determined by several trade-offs: the length of a submessage has to be multiple of the length of the whole message, but very small submessages imply a larger number of cycles to compute the signature. Conversely, long submessages require more LUT storage, which causes energy and area overheads.

Experimentally, it has been determined that subblocks of size 8 bytes signed with eight 1-KB LUTs incur in small time and energy overheads, as shown in Section 4.3. The average command that updates constants modifies 16 values. A subblock of length 8 bytes corresponds to 2 of those values and, therefore, computing the signature for the average constant input data requires 8 cycles. Regarding primitives, the size of the data of an attribute is 48 bytes, which correspond to 3 vertices defined by four 4-byte components each. The average number of attributes per primitive is 3 and, thus, computing the signature for the average primitive requires 18 cycles.

4.2.3 Transaction Elimination

Transaction Elimination (TE) [41] is a technique that reduces main memory bandwidth by avoiding the flush of the Color Buffer in tiles that have the same color as in the preceding frame. Since the reuse distance of two tiles is an entire frame, tile equality is not performed by comparing the colors of all the pixels of a tile but rather signatures of those colors. Whenever a tile has finished being rendered, its colors (the contents in the Color Buffer) are hashed into a signature and compared to the signature of the same tile for the previous frame. If the two signatures are equal, the newly generated colors are not written into the Frame Buffer. Although the exact details of this technique in commercial systems are not fully disclosed, we have modified our cycle-accurate simulator to model an efficient implementation and compare it with our proposed approach. Figure 4.9 presents the extra hardware added in the pipeline to perform Transaction Elimination.

Transaction Elimination is evaluated by only considering the energy overheads caused by the Signature Buffer and the Compute CRC unit, but without taking into account their execution time overheads: while the number of accesses are counted to report energy, it is ideally assumed that the signature for a Color Buffer does not require any execution cycles.

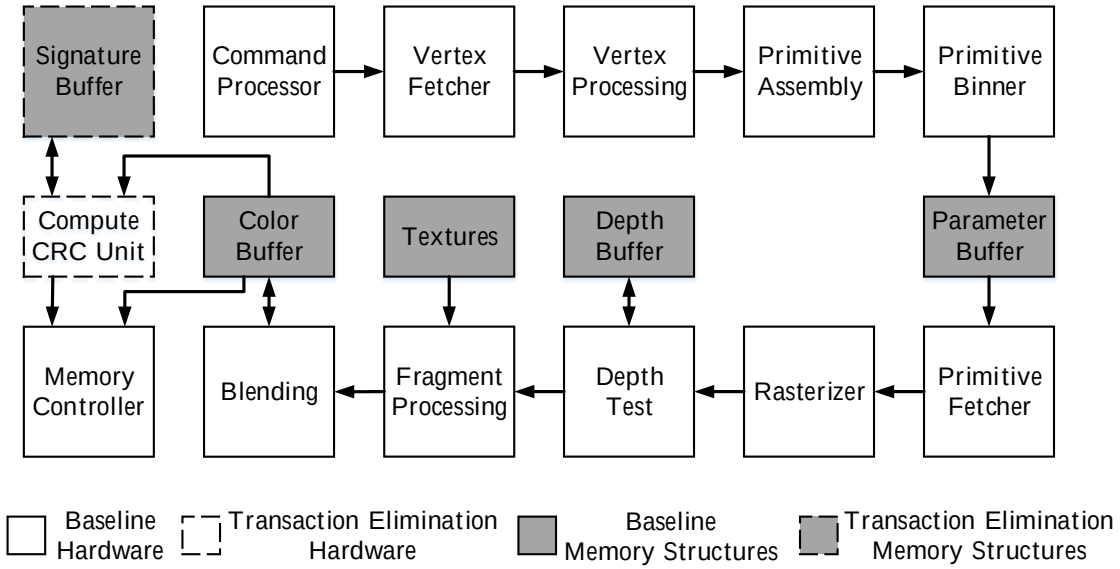


Figure 4.9: Graphics Pipeline including TE.

4.3 Experimental Results

This section presents the main results of Rendering Elimination over the baseline architecture, described in Table 3.1. The parameters employed during the simulations are summarized in Table 4.1. For comparison purposes, Transaction Elimination [41] and Fragment Memoization [8] are also evaluated.

Table 4.1: Parameters considered in the experiments for the structures of Rendering Elimination, Transaction Elimination and Fragment Memoization.

Rendering Elimination	
Compute CRC Unit	12 KB (8 Sign LUTs, 4 Shift LUTs)
Accumulate CRC Unit	4 KB (4 Shift LUTs)
Signature Buffer	3588 entries/frame, 2 frames of signatures, 4 bytes/entry
Overlapped Tiles Queue	128 entries, 12 bits/entry
Constant Bitmap	3588 entries, 1 bit/entry
Transaction Elimination	
Compute CRC Unit	12 KB (8 Sign LUTs, 4 Shift LUTs)
Signature Buffer	3588 entries/frame, 2 frames of signatures, 4 bytes/entry
Fragment Memoization	
Frames rendered in parallel	2
Signature size	32
LUT Number of ways	4
LUT Number of sets	2048

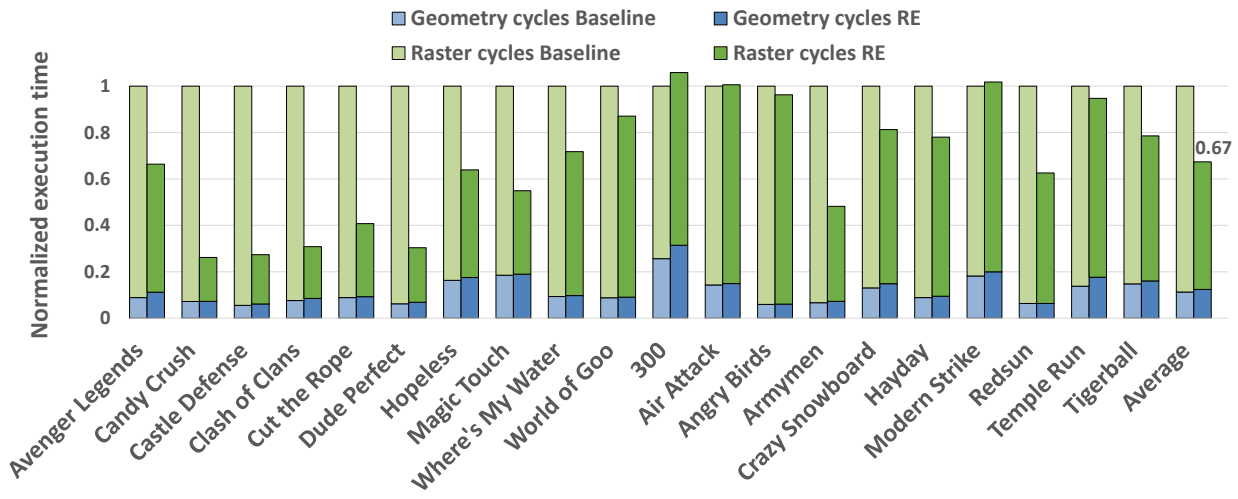


Figure 4.10: Execution cycles of Rendering Elimination (RE) compared to the Baseline GPU.

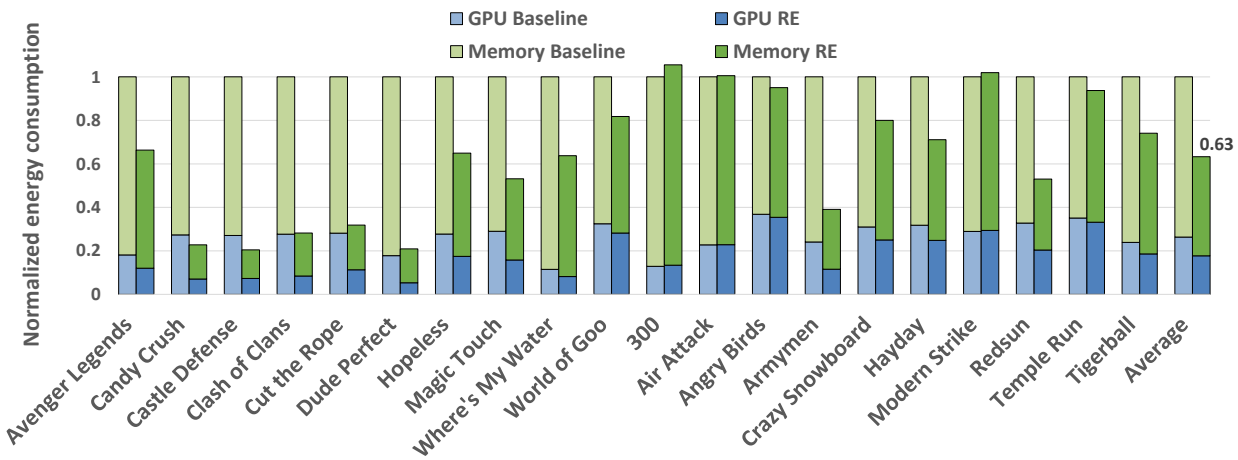


Figure 4.11: Energy consumption of Rendering Elimination (RE) compared to the Baseline GPU.

Figure 4.10 shows execution cycles of Rendering Elimination for the set of benchmarks. The total cycles are normalized to those of the Baseline and divided into cycles corresponding to Geometry and Raster Pipelines. RE achieves an average execution time reduction of 33%, yielding reductions of up to 85% (*Candy Crush*). The execution of the Raster Pipeline using RE is 1.5x faster than the Baseline GPU on average, with maximums of more than 4.5x. On the other hand, the overheads introduced by the technique are almost negligible, since the signature computation is overlapped by previous Geometry Pipeline stages. The pipeline is only stalled when computing signatures for primitives that cover a large amount of tiles, resulting in an overflow of the Overlapped Tiles Queue. These kind of primitives are rare, as can be seen by the fact that, on average, only an additional 2% of geometry cycles are introduced. The overhead of comparing the signatures is even smaller. Considering that accessing the corresponding Signature Buffer entry and performing a simple comparison takes a few cycles while skipping the entire Raster Pipeline can save thousands, these tiny overheads are more than offset by the large performance gains. Such overheads only

result in performance loss in benchmarks that lack redundant tiles and cannot leverage Rendering Elimination at all. Even in those cases, the performance impact is small, with benchmarks such as *Air Attack* or *Modern Strike* having an overhead of less than 1% of the cycles and *300* showing a slowdown of 5%.

Figure 4.11 shows the GPU energy consumption (considering both static and dynamic) when using RE for the set of benchmarks, normalized to the baseline. The total energy is split into two parts: energy spent by the GPU in accessing main memory and energy spent in other activities. As shown, RE brings about an average 37% reduction of the energy consumed by the system, with a 33% reduction of the energy consumed by the GPU and 40% reduction of the energy consumed by main memory. Moreover, RE provides enormous energy savings for benchmarks such as *Candy Crush*, *Castle Defense* or *Dude Perfect*, reducing 80% of the overall energy consumed by the baseline. In benchmarks that do not take advantage of RE, the energy overheads are smaller than 5%. Regarding area, McPAT reports that the cost of the hardware added (CRC LUTs, Signature Buffer, Overlapped Tiles Queue and bitmap) incurs in less than 1% area overhead.

These reductions in execution time and energy consumption are due to an important number of tiles bypassing the execution of the Raster Pipeline and avoiding their corresponding main memory accesses. Figure 4.12 shows the average percentage of tiles that, across neighboring frames, produce the same color (the sum of bottom and mid bars) and the average percentage of tiles that change colors (top bar). The bottom bar depicts the percentage of tiles that Rendering Elimination avoids rendering, which is, on average 47% of the tiles of a frame and 78% of the total redundant tiles. The mid bar shows the percentage of tiles that despite having different inputs end up with the same color (13%). The top bar presents the percentage of tiles with different inputs and different colors (40%). Note that there is not a single occurrence of a tile that changes the color while maintaining the same inputs. Furthermore, Figure 4.12 reveals two different behaviors for the benchmarks analyzed depending on camera movements. The first category, (*Avenger Legends* to *World of Goo*) is composed of workloads with mainly static cameras, so their scenes contain lots of redundant tiles. The second category (*300* to *Tigerball*) is composed of workloads in which some phases contain highly dynamic camera movements and in other phases the scene is static. It can be seen that there is a strong correlation between the number of detected redundant tiles presented on Figure 4.12 and the speedup and energy savings reported in Figures 4.10 and 4.11.

We refer to the above event, where the signature of two tile inputs does not match but the final color of their pixels remains unchanged, as *false negatives*. False negatives do not generate errors, but reveal a broader potential for tile reuse that RE is not capable to detect. On the other hand, since tile inputs are compared using the result of a hash function, there exists the possibility of collisions or *false positives*: pairs of different tile inputs that are mapped to the same signature. A false positive means that the GPU incorrectly reuses a tile that has actually changed in the current frame. However, the probability of such an event with a CRC32 signature is roughly one every 4 billion tiles, i.e., less than one tile per million frames (more than 4 hours playing). Moreover, it would be extremely difficult, or impossible, to spot the incorrect tile by a human, since it would last for only a single frame (less than 20 ms), and it would probably appear very similar to the correct tile due to frame coherency. Actually, zero false positives were found when rendering these benchmarks with CRC32 as a signature.

Eliminating redundant tiles not only reduces the activity of the GPU but it also eliminates all

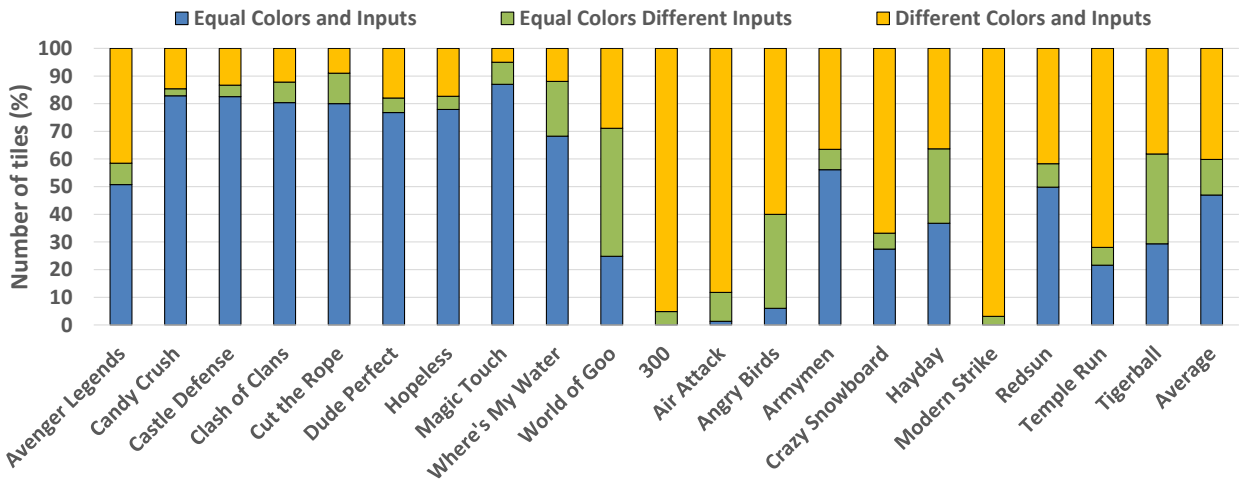


Figure 4.12: Tiles with equal color and inputs, equal color and different inputs, and different color and inputs across neighboring frames.

the associated memory accesses. Figure 4.13 plots the amount of main memory traffic generated by the Raster Pipeline, normalized to the baseline. The total traffic is split into three parts: accesses generated by the Tile Cache when reading primitives from the Parameter Buffer, accesses generated by the Texture Cache when fetching textures in the fragment shaders and accesses generated by flushing the on-chip Color Buffer to the Frame Buffer. As it is shown, Rendering Elimination achieves a significant drop in traffic to main memory (44% on average).

4.3.1 Rendering Elimination compared to Fragment Memoization and Transaction Elimination

Figure 4.14 compares the benefits in execution time and energy consumption of Rendering Elimination over Transaction Elimination (TE) and Fragment Memoization. The TE implementation uses the same hardware structures as Rendering Elimination (see details in Section 4.2.3). Fragment Memoization is modelled as originally proposed in the work of Arnau et al. [8], executing 2 frames in parallel and using a 32-bit hash that discards the screen coordinates, but their default 512-entry 4-way LUT has been augmented to 2048 entries to better compare to the chip area of RE.

TE avoids only the Color Buffer flushes to main memory, while RE bypasses the whole Raster Pipeline execution for redundant tiles. Therefore, while TE reduces by a 9% the energy consumption with respect to the baseline GPU, RE outperforms it and achieves a reduction of 37%. Note that in benchmarks with a large percentage of redundant tiles such as *Castle Defense*, RE achieves an additional 60% energy savings compared with TE. Moreover, since the flush of the Color Buffer represents a relatively small portion of the total time of the Raster Pipeline, RE far surpasses the performance benefits of TE.

Rendering Elimination also provides a significant improvement when compared to Fragment Memoization. One would expect that, by working at a fragment granularity, Memoization could discover more redundancy than a technique working at tile level. However, such granularity also

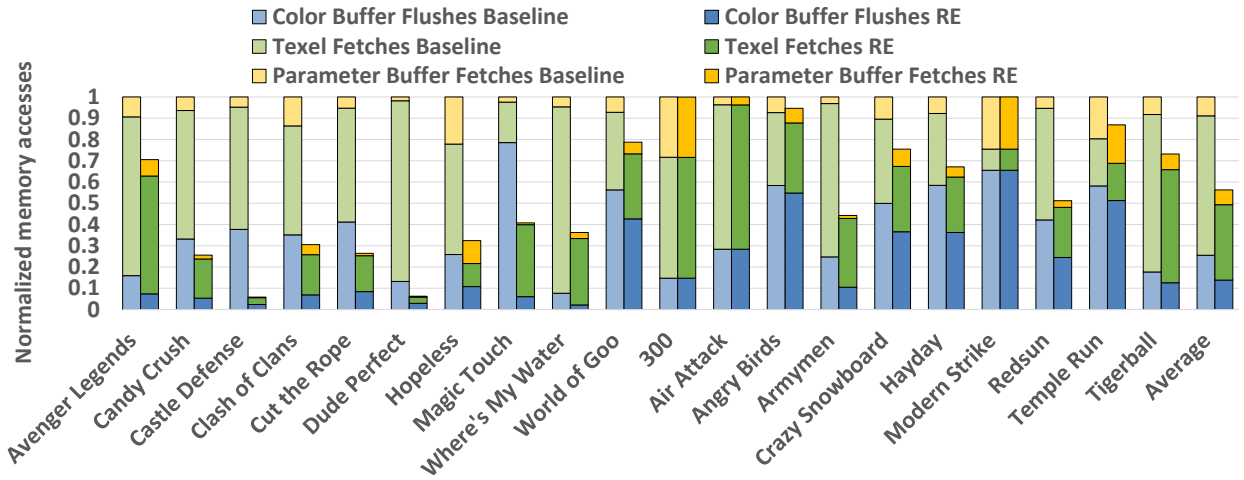
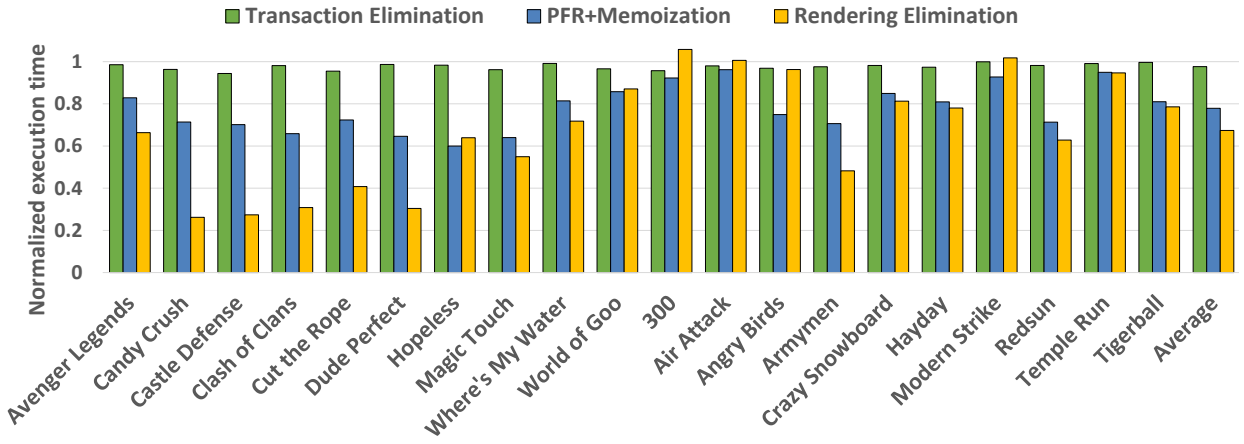


Figure 4.13: RE memory bandwidth compared to baseline: Parameter Buffer and Texel fetches and Color Buffer flushes.

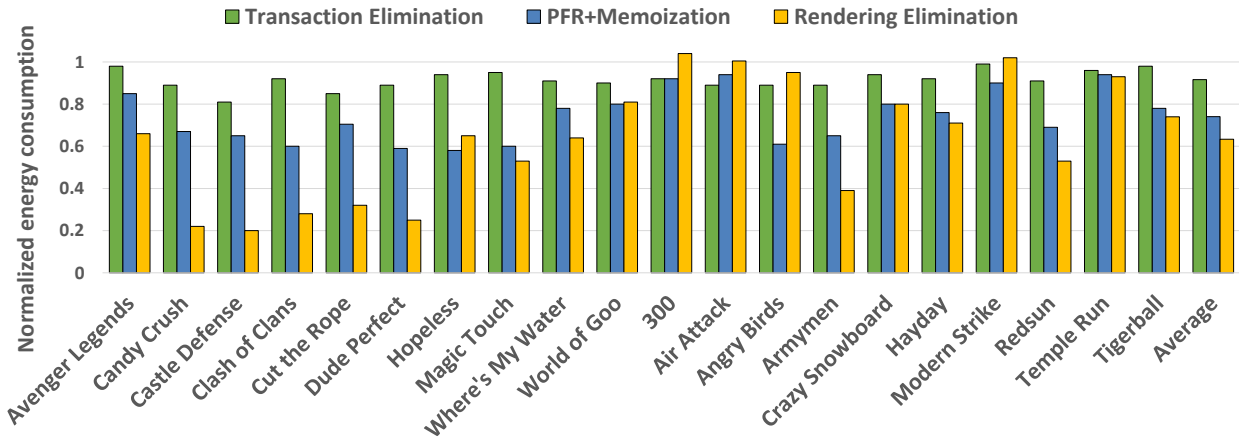
requires a bigger storage and, as already pointed out in the original paper, a realistic space-limited LUT only captures on average 60% of that potential, whereas RE captures all of the redundant tiles with equal inputs, discarding all their corresponding activity and yielding an average of 10% more energy and execution time savings than Memoization.

The only notable exceptions are *Hopeless* and *World of Goo*, two applications containing rather rare cases in which a significant portion of the screen is black and, therefore, the pressure on the LUT storage is heavily reduced by being able to render the scene with a small number of repeated fragments. Moreover, because of the large reuse distance between redundant fragments, Fragment Memoization requires significant modifications in the pipeline to enable rendering of multiple frames in parallel. While executing two frames in parallel has benefits beyond Memoization, it has two major drawbacks that RE does not. First, it implies a significant re-design of the whole GPU. Second, it generates input response lag because of the parallel frame rendering process. To alleviate this side effect it must be disabled during frames where the user introduces inputs.

Transaction Elimination and Memoization may also obtain savings for benchmarks in which Rendering Elimination cannot. As Figure 4.12 presents, there is a subset of tiles whose rendering outputs the same color as in the preceding frame but do not have the same inputs as in the preceding frame (depicted in the mid bar). On average, this occurs for 12% of the tiles. This phenomenon may occur, for instance, when the only differences between the two tiles happen on occluded fragments that are eventually culled by the Depth Test and do not contribute to the final color of the tile, or for scenes with quick camera panning movements where most of the background texture contains a single plain color. Consequently, in benchmarks where RE detects a small percentage of equal tiles, such as *Angry Birds* or in benchmarks where RE does not detect any equal tiles, such as *300*, TE and Memoization may obtain better energy savings than RE.



(a) Execution cycles.



(b) Energy consumption.

Figure 4.14: Comparison of Transaction Elimination, Fragment Memoization with Parallel Frame Rendering (PFR) and Rendering Elimination against the Baseline GPU.

4.4 Conclusions

This chapter has presented Rendering Elimination (RE), a novel micro-architectural technique for Tile-Based Rendering GPUs that effectively reduces shading computations and memory accesses by means of culling redundant tiles across consecutive frames. Since RE detects a redundant tile before it is dispatched to the Raster Pipeline, the entire computation (which includes rasterization, depth test, fragment processing, blending, etc.) is avoided, as well as all the associated energy-consuming memory accesses to the Parameter Buffer, Textures and Frame Buffer.

Results show that RE outperforms state-of-the-art techniques such as Transaction Elimination or Fragment Memoization, which are only able to bypass a single pipeline stage. Compared to the baseline GPU, RE achieves an average execution time reduction of 33% and reduces the GPU and main memory energy consumption by 33% and 40%, respectively. The hardware overhead of RE

is minimum, requiring less than 1% of the total area of the GPU, while its latency is hidden by other processes of the graphics pipeline. In terms of energy, RE incurs a negligible overhead of less than 0.5% of the total GPU energy. RE is especially efficient in benchmarks with small camera movements, with execution time reductions as high as 86% and energy savings up to 80%. Even in benchmarks without any significant amount of redundant tiles, the performance impact is small.

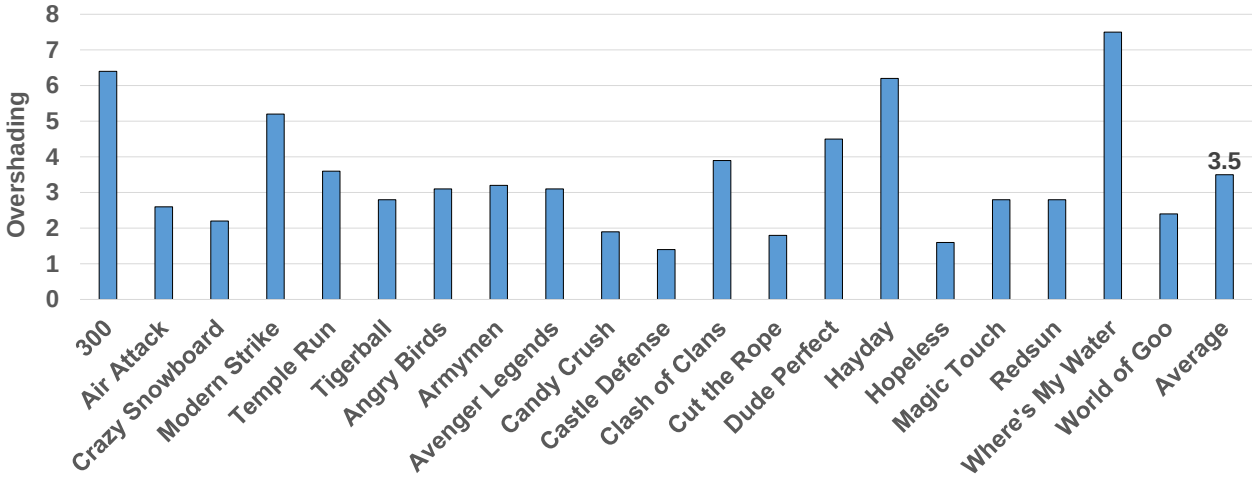
5

Early Visibility Resolution

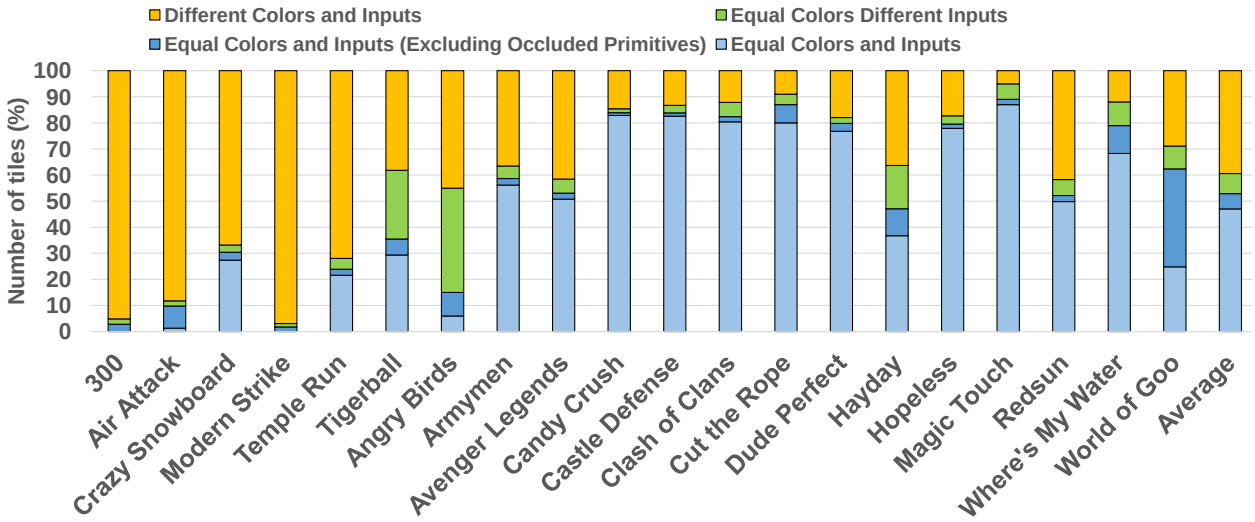
This chapter presents Early Visibility Resolution, a mechanism that leverages the visibility information obtained in a frame to speculatively determine the visibility in the following one much earlier in the pipeline than traditional approaches.

When rendering a scene, it is normal that multiple objects overlap at each pixel and, consequently, that the color for a pixel is computed multiple times, a phenomenon known as *overshading*. If some of those objects are opaque, then a significant amount of processing is devoted to computing colors that will not be visible in the final image. Figure 5.1a shows the overshading factor for several commercial Android applications. It can be seen that all applications shade their pixels more than once, with the average application doing so 3.5 times and several applications having an overshading factor larger than 5. Early Visibility Resolution 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 Depth Test.

Additionally, in some cases hidden primitives may reduce the amount of work saved by Rendering Elimination, the mechanism described in Chapter 4. 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. Figure 5.1b shows the average percentage of tiles that, across consecutive frames, change colors (top bar) and the average percentage of tiles that produce the same colors (the sum of the three bottom bars). The bottom bar represents the percentage of tiles whose processing can be avoided by Rendering Elimination, while the bar second from the bottom indicates the additional tiles that can be detected as equal if occluded primitives are not included in Rendering Elimination's tile signature. It can be seen that by combining Rendering Elimination with Early Visibility Resolution, the amount of skipped tiles increases by 12%, detecting 87% of the overall redundant tiles.



(a) Overshading factor (number of shaded fragments per pixel).



(b) Tiles with equal color and inputs, equal color and inputs whenever occluded primitives are not considered, equal color and different inputs, and different color and inputs across neighboring frames.

Figure 5.1: Effects of the baseline visibility resolution in the Graphics Pipeline.

5.1 Early Detection of Occluded Primitives

Detecting occluded primitives early in the pipeline can prevent their processing and avoid a significant amount of ineffectual work. However, as visibility resolution is a complex problem, the proposed solution relies on a simplification that estimates it with a low implementation cost. The similarity between consecutive frames entails that visibility tends to remain constant: 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, primitives are labelled 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. As shown in Section 5.2, this visibility prediction scheme is implemented so that redundant activity is reduced while maintaining the correctness of rendered images: the reordering of primitives predicted to be occluded is only applied to the ones that use the Depth Test to resolve their visibility, which avoids rendering errors in case of a misprediction. Additionally, removing primitives predicted to be occluded from Rendering Elimination's signature does not generate rendering errors, since changes in visibility require changes in the attributes of one or more primitives predicted to be visible, which results in different signatures across frames.

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 [46], 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, a distinction must be made between primitives that write on the Z Buffer (WOZ primitives) and primitives that do not ($NWOZ$).

5.1.1 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 Primitive Binning stage (vertex depths and identifiers of the tiles the primitive overlaps), 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.

5.1.2 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, the number of different draw commands that have produced primitives that overlapped each particular tile is tallied and stored 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, those identifiers can be used 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.

5.1.3 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.

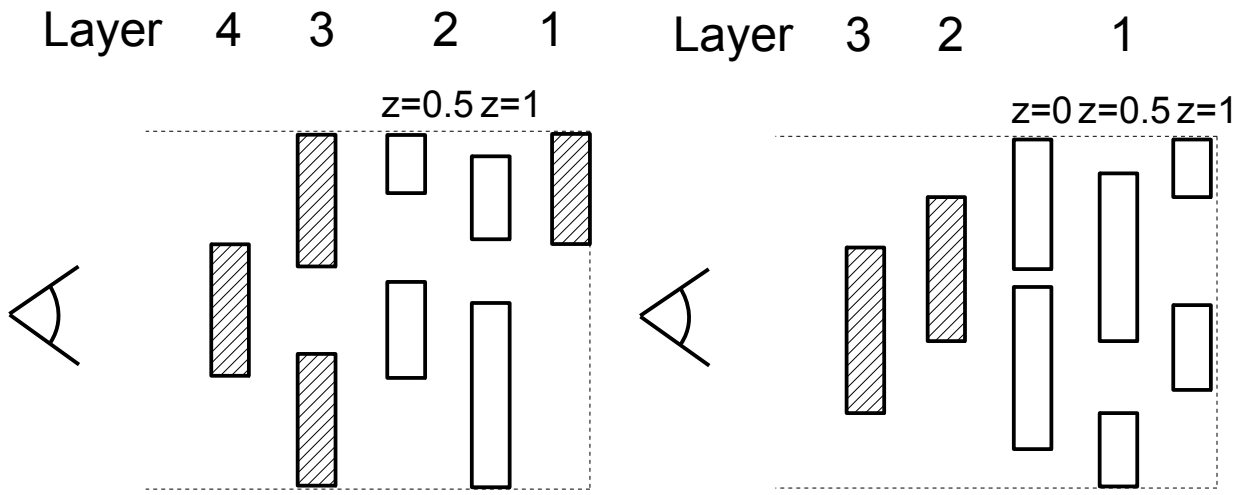
WOZ primitives are also assigned a layer identifier to help compare its age relative to NWOZ primitives. However, since resolving visibility among WOZ primitives themselves is not determined by their relative age but by comparing their explicit depth values, we can assign the same layer identifier to 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}, the type of primitive it belongs to is determined and the proper FVP depth value for the tile (either Z_{far} or L_{far}) is stored. A boolean value termed *FVP-type* is then set to indicate whether the stored FVP corresponds to a WOZ or a NWOZ primitive.

Figure 5.2 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.

In the scenario presented in Figure 5.2a, 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.

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



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

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

Figure 5.2: FVP depth computation in tiles with both WOZ primitives (white) and NWOZ primitives (striped).

In the scenario presented in Figure 5.2b, 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}.

5.2 Removing Ineffectual Computations with EVR

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

5.2.1 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.

The speculative visibility determination mechanism described in Section 5.1 can be used 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 Primitive Binning 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 4.

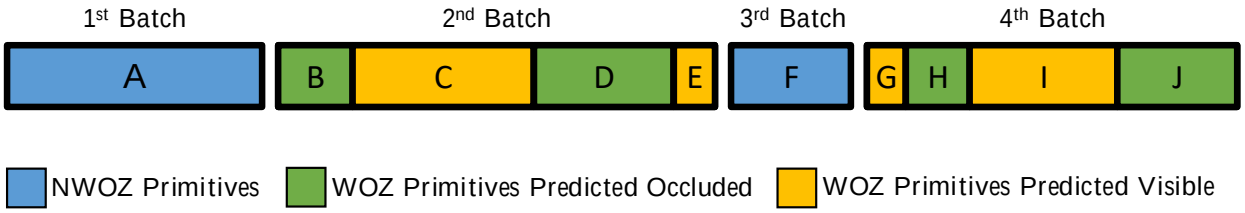
Algorithm 4 Reordering Algorithm based on FVP

```
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
```

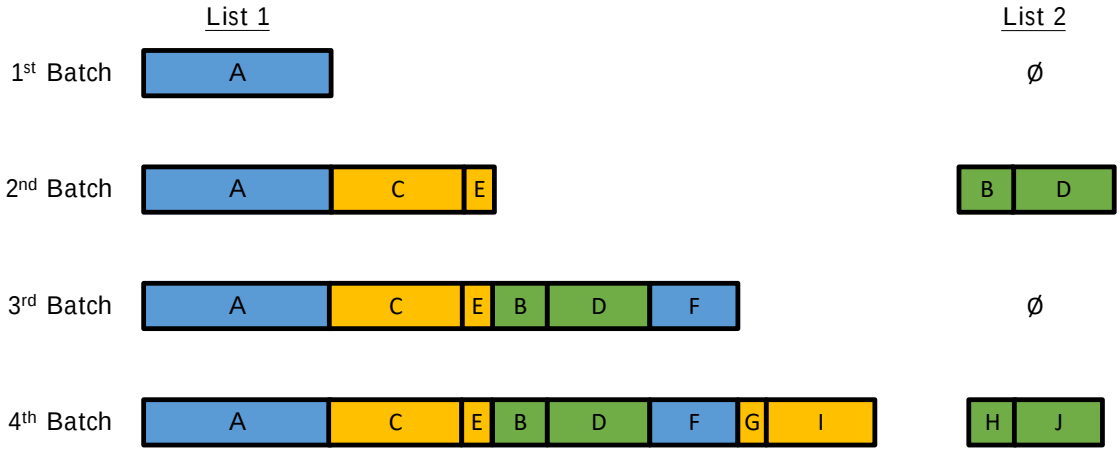
This algorithm only reorders opaque WOZ primitives among themselves, while preserving the order of NWOZ primitives against themselves and against WOZ primitives. Reordering WOZ primitives does not incur in rendering errors: NWOZ primitives are not reordered and all WOZ

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.

Figure 5.3a illustrates the behavior of Algorithm 4 by 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 5.3b shows how they are reordered.



(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 5.3: Reordering algorithm example.

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.

5.2.2 Rendering Elimination Improvement

Rendering Elimination, described in Chapter 4, 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. The approximate visibility resolution mechanism described in Section 5.1 can be used to compute signatures only with visible primitives so as to improve the effectiveness of Rendering Elimination's tile redundancy detection.

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 5.1, Rendering Elimination can be extended as follows. For each sorted primitive, the depth of its closest-to-the-observer vertex 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 5.1 presents the four possibilities regarding the resolved visibility of a primitive (either visible or occluded) across two consecutive frames.

Table 5.1: Visibility casuistry

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

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.

5.3 Implementation

This section describes 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 5.4 shows how they are integrated into a TBR GPU.

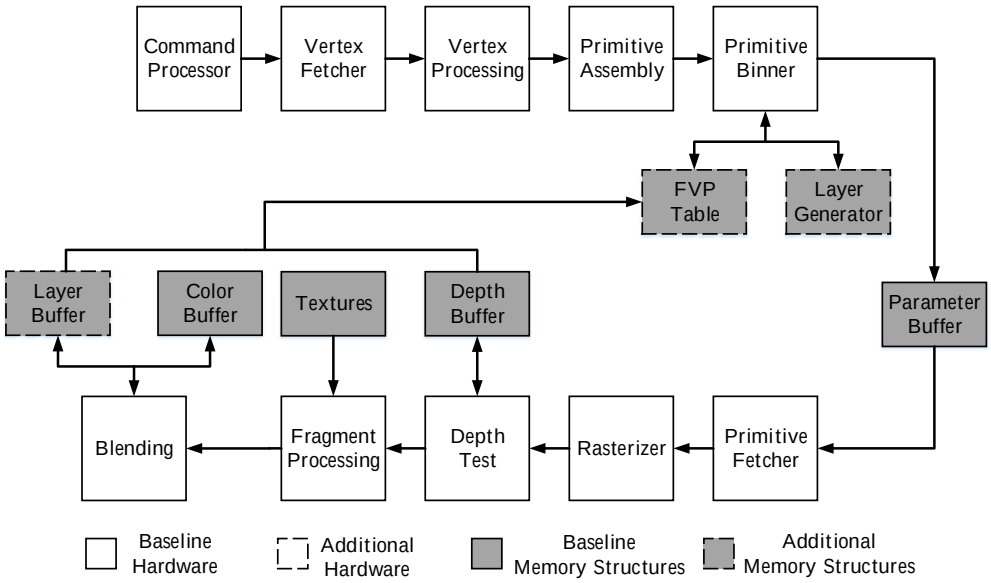


Figure 5.4: Graphics Pipeline including the structures needed to implement FVP computation.

5.3.1 Layer Generator Table

As discussed in Section 5.1, 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 (Drawcalls) 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.

A small, on-chip LUT named *Layer Generator Table* is employed 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) layers can be assigned to every primitive in all the tiles it overlaps during the Primitive Binning 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. Section 5.4 shows that minor energy overheads are produced by storing layer identifiers this way and generating additional accesses to Main Memory.

The command and layer identifier fields have been dimensioned based on common performance guidelines for mobile graphics applications. As it is advised that applications do not make hundreds of drawcalls [20, 28], 12 bits are devoted to store command identifiers, supporting 4096 unique drawcalls per frame. The worst case in which primitives from all commands can overlap a tile

is assumed and, consequently, layer identifiers are also comprised of 12 bits. The LGT has been modelled as a McPAT SRAM to obtain its power and area.

5.3.2 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) an on-chip buffer can be used to keep track of per-pixel information for an entire tile. This buffer, called *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. The same alpha value that fragments employ to blend with colors previously written in the Color Buffer is used to detect opacity. 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. McPAT components have been used in order to obtain an area and power estimation of the hardware necessary to keep track of the visible layers and the FVP-type: an SRAM to model the Layer Buffer and a D flip-flop for the ZR Register.

5.3.3 FVP Computation and FVP Table

The FVP depth of a tile in the previous frame is used in order to predict if a primitive is likely to be visible, so the entire set of per-tile FVP depths must be stored. Such information is maintained in a structure called *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. The FVP for a tile is stored using 24 bits, since Z values are represented by 24 bits while layers only use 12. Consequently, each FVP Table entry contains 25 bits. The FVP Table has been modelled as a McPAT SRAM to obtain its power and area.

Whenever a tile finishes rendering, Z_{far} and L_{far} values are obtained by computing the maximum value of the Z Buffer and the minimum value of the Layer Buffer, respectively. This computation has been implemented using a comparator tree, a perfect binary tree with height 4, whose root contains the maximum/minimum value of the 16 leafs of the tree, which correspond to a 16-element row of either the Z or Layer Buffer. The nodes of the tree represent a comparator that computes the maximum/minimum of its two children and a register to store the result of the comparison. The comparator tree is, therefore, composed of 2^{l-1} comparators and registers at each level l ($l > 0$),

amounting to 15 comparators and registers for the entire tree. This pipelined design allows the tree to start processing a new Buffer row each cycle, for which its maximum/minimum value is obtained and stored in an additional register after 4 cycles. The temporary maximum/minimum values for all rows are obtained after 19 cycles, which are then fed to the comparator tree. Therefore, 23 cycles are required to obtain the maximum/minimum value for the entire Buffer. The comparator tree and the temporary registers have been implemented in VHDL and synthesized to obtain its delay and power using the Synopsys Design Compiler, the modules of the DesignWare library and the 32/28nm technology library from Synopsys [73].

After computing Z_{far} and L_{far}, the FVP-type for the tile 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.

5.4 Experimental Results

This section presents the main results of Early Visibility Resolution over the baseline architecture, described in Table 3.1, and the improvements over Rendering Elimination, whose parameters are listed in Table 4.1. The parameters employed during the simulations are summarized in Table 5.2.

Table 5.2: Parameters considered in the experiments for the structures of Early Visibility Resolution.

Early Visibility Resolution Structures	
Layer Generator Table	3588 entries, 25 bits/entry
Layer Buffer	256 entries, 12 bits/entry
FVP Table	3588 entries, 25 bits/entry

Figure 5.5 shows the energy consumption of the GPU-Memory system normalized to the Baseline GPU for the 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% (*Castle Defense*, *Dude Perfect*). Figure 5.6 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% (*Candy Crush*, *Castle Defense*, *Dude Perfect*).

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 5.5. Figure 5.5 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

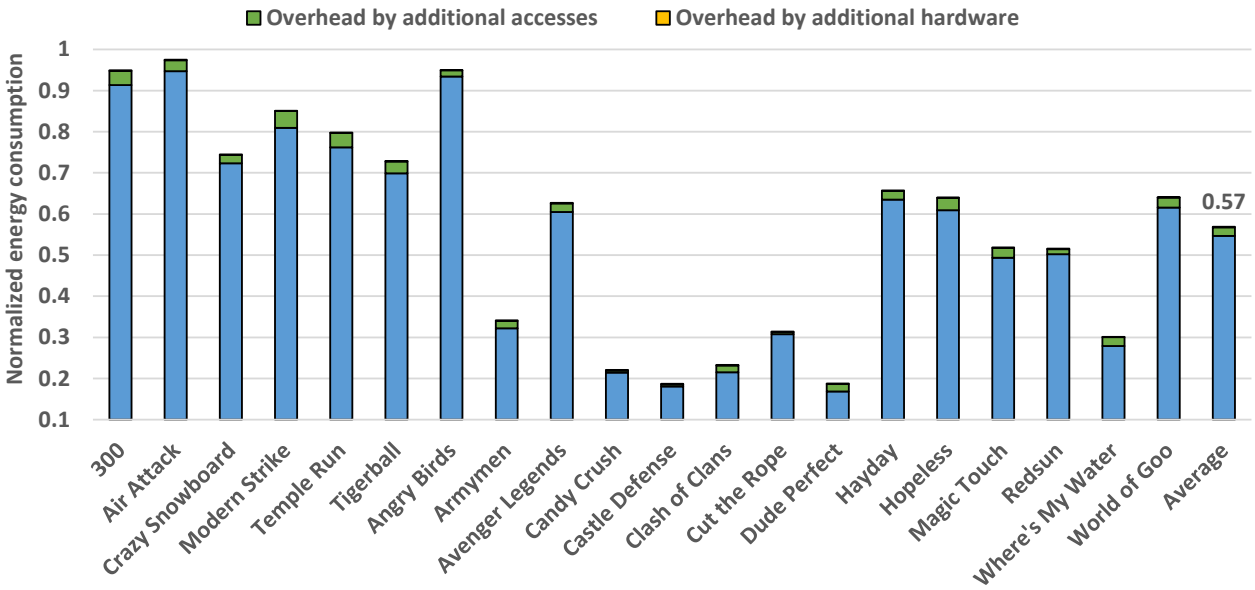


Figure 5.5: Energy consumption of EVR normalized to the Baseline GPU.

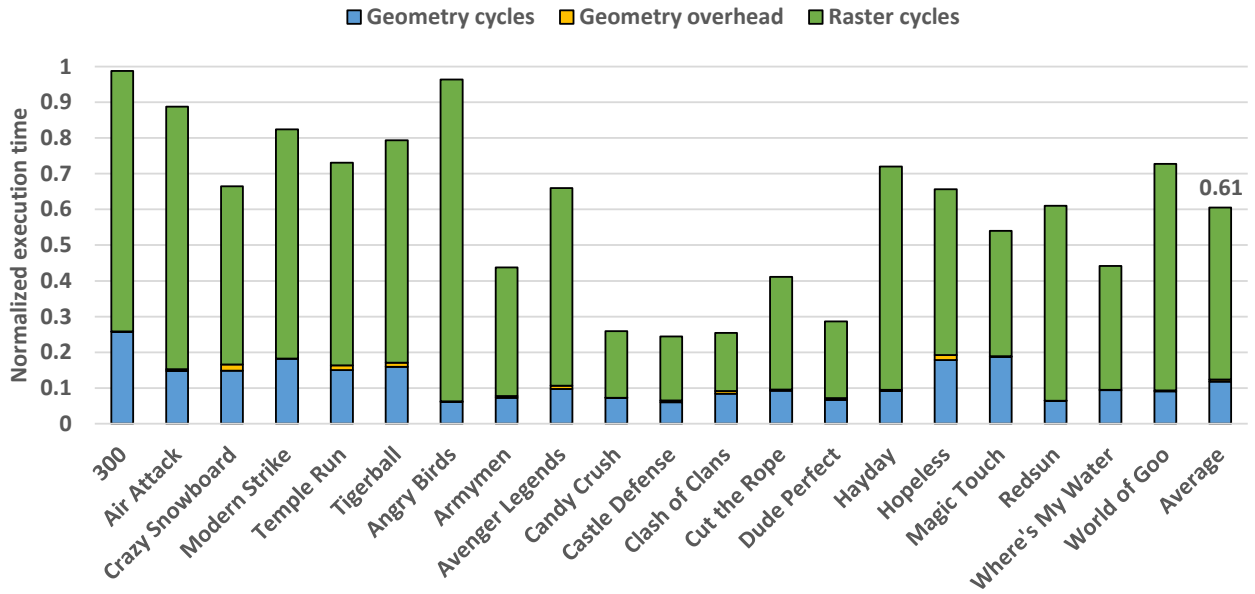


Figure 5.6: Execution time of EVR normalized to the Baseline GPU.

large number of tiles, since the pipeline is stalled waiting for all signatures to be sequentially updated. Figure 5.6 reports such overheads in execution time which, on average, represents 0.5% of the total.

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 5.7 shows the number of fragments shaded per pixel using the proposed Early Visibility Resolution mechanism to reorder primitives (*EVR*) compared to the baseline for

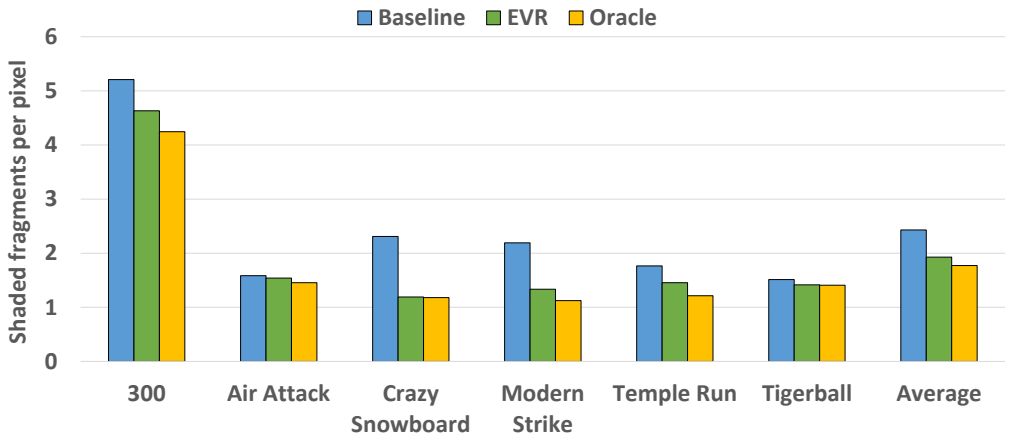


Figure 5.7: Comparison of the number of shaded fragments per pixel among the baseline GPU, Early Visibility Resolution (EVR) and an oracle.

the set of 3D benchmarks. EVR is also compared 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. This oracle approach is equivalent in overshading to the PoweVR Tile-Based Deferred Rendering architecture from Imagination Technologies [58]. The PowerVR architecture performs a Hidden Surface Removal step, in which the primitives in a tile are rasterized only for position and depth and the resulting fragments are Depth Tested. Once Hidden Surface Removal is complete, the Depth Buffer contains in each position the depth of the closest opaque fragment. The primitives of the tile are then fully rasterized and processed, with only visible fragments passing the Early Depth Test and being shaded. EVR significantly reduces (20%) the number of vainly shaded fragments, and its results are close to those obtained by an oracle approach without any need to perform any Hidden Surface Removal pass. 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 5.8 shows the percentage of detected redundant tiles –producing the same colors as the previous frame– for the 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.

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 *Modern Strike*. 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

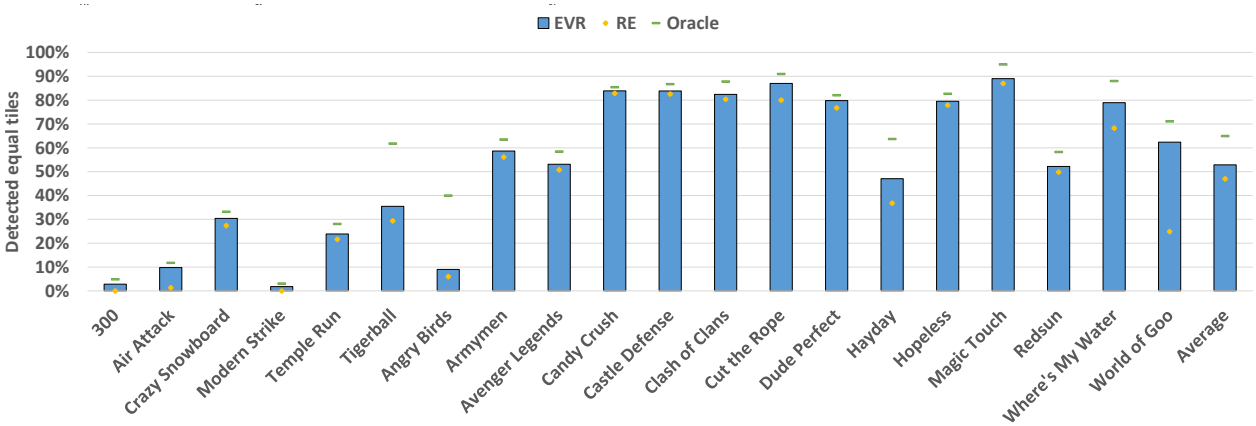


Figure 5.8: Percentage of detected equal tiles by Early Visibility Resolution (EVR), compared to Rendering Elimination (RE) and an oracle.

with a high degree of redundancy detected by RE, such as *Castle Defense* or *Magic Touch*. In some applications such as *Hayday* or *Where's My Water*, the additional tiles eliminated exceeds 10%, with a maximum of 30%.

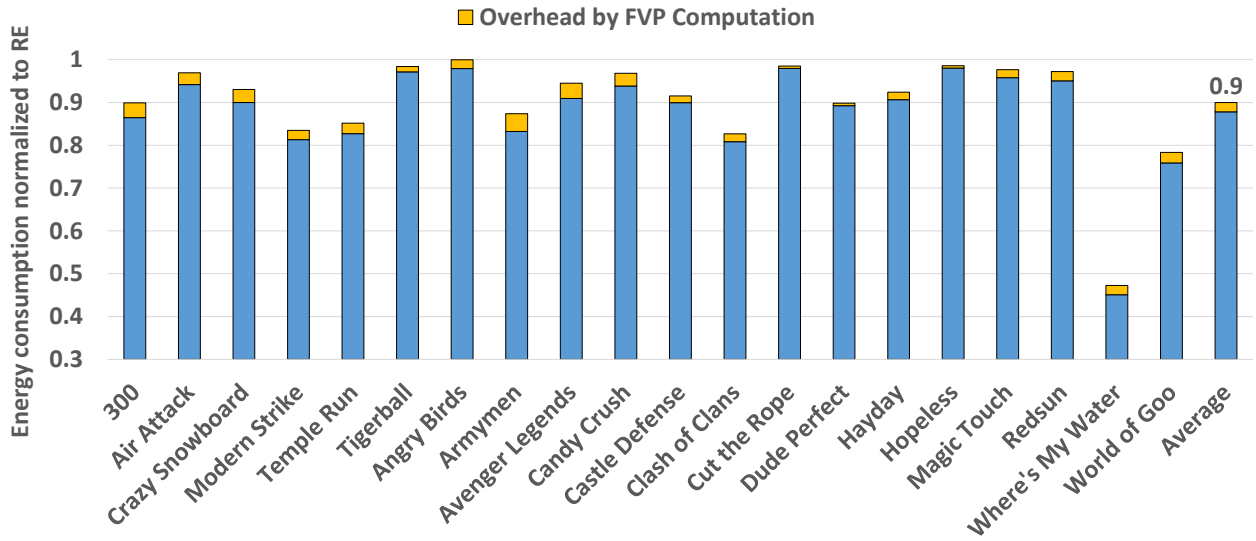


Figure 5.9: Energy consumption of Early Visibility Resolution (EVR) normalized to Rendering Elimination (RE).

The additional redundant tiles detected results in an average 10% reduction of energy consumption compared to the baseline RE, as presented in Figure 5.9. 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 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

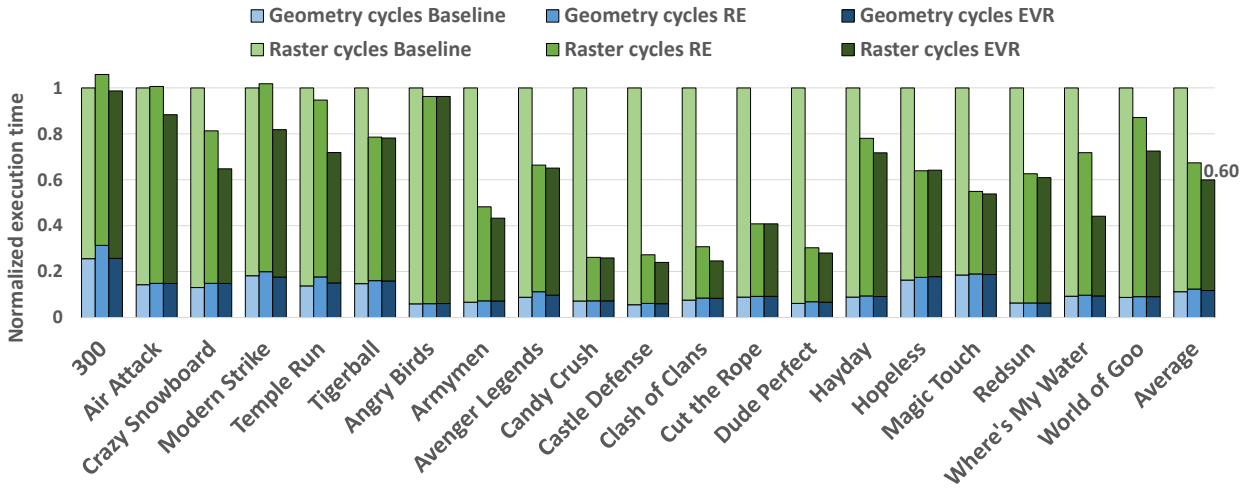


Figure 5.10: Execution time comparison of Early Visibility Resolution (EVR) and Rendering Elimination (RE) against the Baseline GPU.

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 5.10, which compares execution time, broken down into Geometry and Raster Pipelines, for the Baseline GPU, RE and the proposed Early Visibility Resolution approach. 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 EVR 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 the EVR scheme does not reduce Geometry cycles is *Hopeless*, 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, adding Early Visibility Resolution reduces the execution time in the Geometry Pipeline by 4% with respect to the baseline Rendering Elimination.

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 *Modern Strike*. On the other hand, EVR 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.

5.5 Conclusions

This chapter has 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, the information of a frame is used to estimate the visibility for the following one.

The proposed technique collects the depth of the farthest visible fragment (FVP) of every tile whenever its rendering process is complete. For each overlapped tile, the depth of the closest vertex of a primitive is compared against the FVP depth stored for that tile in the preceding frame. If it is farther, the primitive is predicted to be occluded.

This chapter demonstrates the benefits of early visibility prediction to remove ineffectual computations, by increasing the effectiveness of the Early Depth Test, a commonly used technique in contemporary GPUs, and Rendering Elimination, the technique presented in Chapter 4 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.

The proposed 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.

6

Dynamic Sampling Rate

This chapter presents Dynamic Sampling Rate, a hardware mechanism that analyzes the spatial frequencies of the scene once it has been rendered. Then, it leverages the temporal coherence in consecutive frames to decide, for each region of the screen, the lowest sampling rate to employ in the next frame that maintains image quality.

6.1 Sampling Rate Estimation

6.1.1 Frequency Analysis

In real-time rendering, triangles are usually discretized into fragments by sampling them at the center of each pixel, hence at a rate of once per pixel. According to the Nyquist Sampling Theorem [53], that sampling rate allows capturing changes in the image every two pixels or more. However, not all tiles require such a sampling rate, because not all parts of the screen contain high frequencies, or changes in the image in a short space.

Figure 6.1 illustrates this phenomenon by comparing two different regions of the screen in a given frame: while 6.1c contains significant level of detail, 6.1b is homogeneous with a single color and, therefore, does not require per-pixel sampling. Figure 6.2 quantifies, for a variety of mobile graphics applications, the number of 16x16-pixel regions of the screen that do not contain enough level of detail for them to require one sample per pixel. It shows that on average almost half of the screen can be processed at a lower sampling rate without affecting image quality. Therefore, a (much) lower sampling rate may be enough to represent the original signals. Properly identifying and removing the large amount of resources devoted to these unnecessary computations can lead to a substantial reduction in energy consumption.

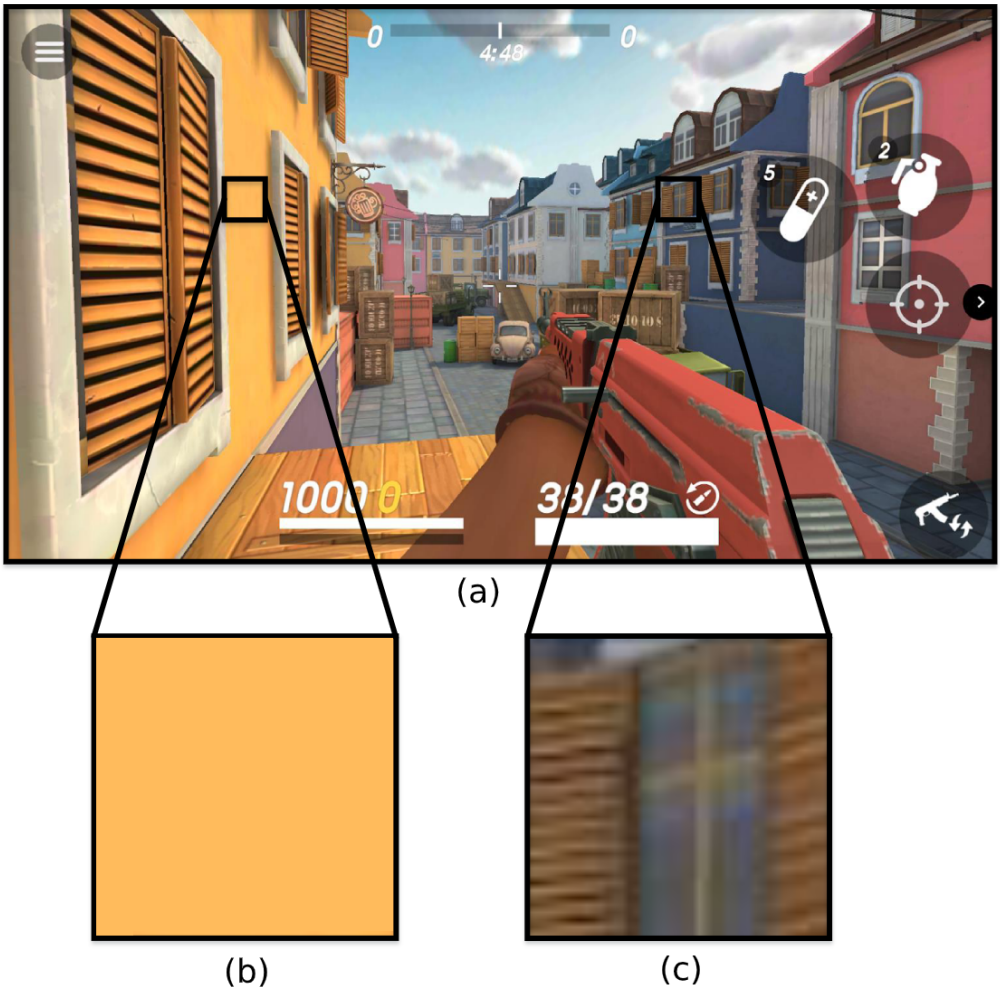


Figure 6.1: Difference in level of detail across a frame. a) Frame of the game Guns of Boom. b) Region with low level of detail. c) Region with significant level of detail.

A well known method to obtain the frequency components of an image is the Discrete Cosine Transform (DCT) [1]. As a Fourier-related transform, the DCT maps a function (an image) from the spatial domain to a set of coefficients of basis functions localized in the frequency spectrum. Those basis functions correspond to sinusoids of a certain frequency and are visually represented in Figure 6.3. It can be seen that as either the x or y axis increase, the basis function is a sinusoid with higher variation rate, i.e., with higher frequency. Applying a 2D DCT to a block of NxN pixel colors results in a NxN matrix of values, the coefficients of the linear combination of basis functions which represent the original image in the frequency domain. The coefficient present in each element of the matrix indicates how much of that particular frequency is found in the original image.

The 2D DCT has several characteristics that make it an ideal choice for the type of real-time frequency analysis required to find the optimal sampling rate for a tile:

- It assumes an even symmetry of the function: by construction, the image is mirrored in all its borders, which avoids artificial high frequency components that other transforms (such

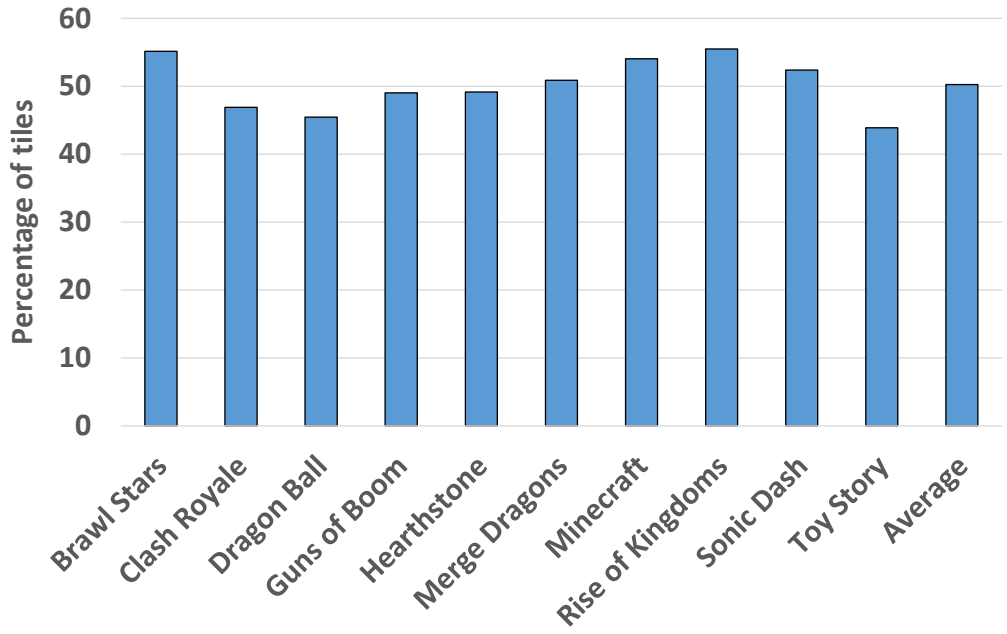


Figure 6.2: Number of 16x16 tiles that can be sampled at a rate lower than one sample per pixel without generating per-tile visible artifacts. Section 6.3 describes the methodology employed for this categorization.

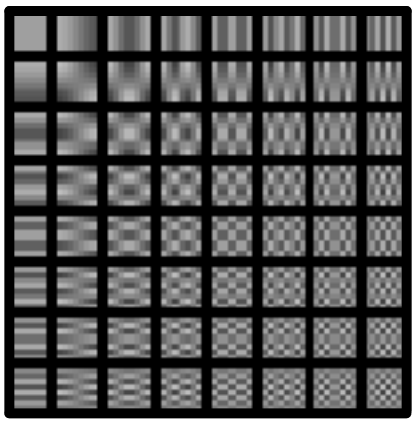


Figure 6.3: DCT basis functions for N=8 pixels.

as the Discrete Fourier Transform) introduce by only considering a NxN pixel subset of the image.

- It has very high energy compaction, which means that the great majority of frequency information is summarized in the upper-left region of the result matrix. This allows us to make sampling rate decisions only considering a subset of the NxN coefficients.
- It has a low complexity cost in comparison with other transforms as only cosines are computed.

Additionally, the 2D DCT is a separable function, which allows for the linear computation of all

the elements in one dimension followed by the linear computation of all the elements in the second dimension [60]. These characteristics allow us to implement a fast and energy efficient hardware unit to analyze the frequency components of a tile, explained in more detail in Section 6.4.

6.2 Dynamic Sampling Rate

This section describes how the 2D DCT is used to estimate the optimal sampling rate for a tile, i.e., the lowest sampling rate that does not introduce visible artifacts in the overall frame, and how to dynamically adapt it to image changes over time.

When the rendering process of the tile finishes, the Color Buffer contains the final color for all the pixels of the tile. A small hardware unit is added to the end of the pipeline to take these colors as inputs and compute their 2D DCT. Then, it analyzes the resulting matrix of coefficients to determine if the current sampling rate for the tile is optimal. All the DCT coefficients are first aggregated into a single value that summarizes the amount of high-frequency information of the tile. Although a plethora of metrics exist, it was empirically determined that the maximum absolute value among the coefficients corresponding to high-frequency diagonals suffices, which will be referred to it as *MaxC* (the diagonal k is defined as the set of all elements of the matrix whose row index plus column index is equal to k: for instance, diagonal 3 consists of elements $(0,3), (1,2), (2,1)$ and $(3,0)$). The rationale under this choice is that, intuitively, a high sampling rate is dependant on whether the largest high-frequency component is big enough rather than the effect of multiple high-frequency components combined. The low-frequency components of the matrix are not taken into account in the computation of *MaxC*.

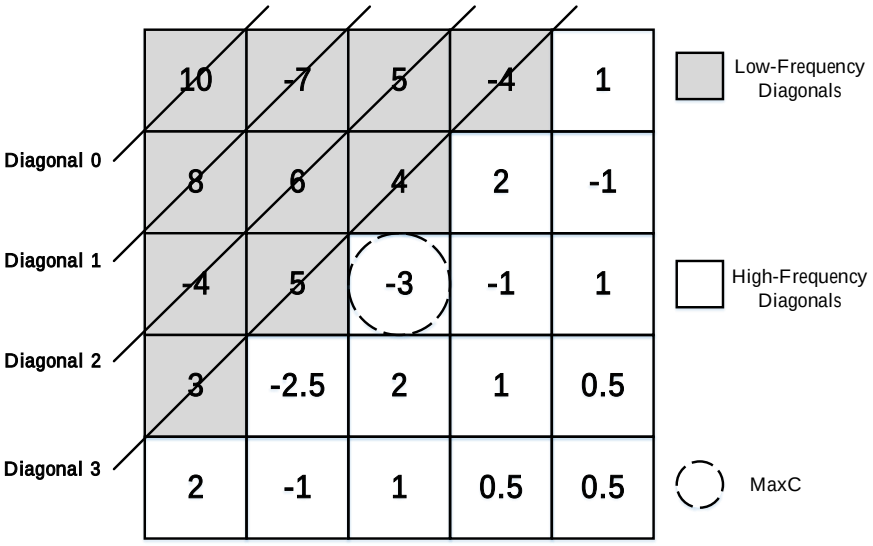


Figure 6.4: MaxC determination example.

Figure 6.4 illustrates the determination of *MaxC* in a 5x5 coefficient matrix in which diagonals 0 through 3 are considered as low-frequency diagonals. As shown, *MaxC* is 3, since it is the highest absolute value among all the high-frequency diagonals. Although larger values appear in

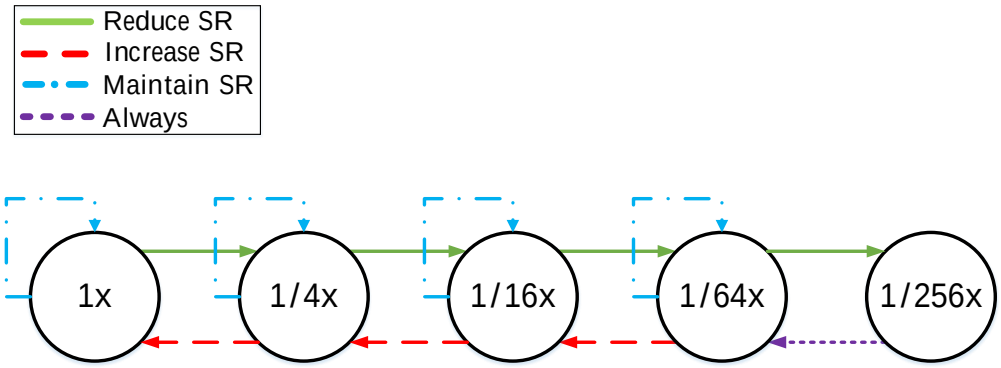


Figure 6.5: Dynamic Sampling Rate Finite-State Machine.

the low-frequency diagonals, they are ignored.

Then, a simple test is conducted to decide the new sampling rate for the tile: *MaxC* is compared against two different thresholds.

- The first threshold, labelled *Reduce Threshold* (T_R), represents the maximum frequency a tile can contain for it to be sampled at a rate lower than the current one. If *MaxC* is lower than the Reduce Threshold, the sampling rate for the tile is reduced.
- The second threshold, labelled *Increase Threshold* (T_I) represents the maximum frequency a tile can contain for it to be sampled at the current rate. If *MaxC* is greater than the Increase Threshold, the sampling rate for the tile is increased.

In the case that *MaxC* is neither lower than the Reduce Threshold nor greater than the Increase Threshold, the sampling rate for the tile does not change. The new sampling rate for each tile is stored and used to process it in the next frame. Because of the frame coherence property of graphics animations, the frequencies of tiles tend to remain constant across frames, which allows to use the estimated-best sampling rate for a tile in the following frame. The scene is, therefore, not sampled uniformly neither in space nor time: each tile is rasterized with an independent sampling rate and it may be modified across frames to adapt to image changes.

Figure 6.5 shows the FSM that manages the dynamic sampling rate determination. Five different sampling rates are considered: sampling at the center of every pixel (baseline sampling rate) and sampling at the center of every square block of 4, 16, 64 or 256 pixels (as shown in Figure 6.6). These sampling rates will be referred to as 1x, 1/4x, 1/16x, 1/64x and 1/256x, respectively, and are motivated by the baseline GPU architecture employed in this work, which utilizes tiles of 16x16 pixels. Each state in the FSM corresponds to halving the previous sample rate in both X and Y dimensions, and the lowest state only generates one sample per tile. The transitions among states are controlled by the heuristic decision described above, based on a $< T, D >$ tuple that contains: the Thresholds (T) to which *MaxC* is compared to, and the number of low-frequency matrix Diagonals that are ignored (D) for its computation. We label as $< T_R, D_R >$ the tuples for the Reduce transitions and as $< T_I, D_I >$ the tuples for the Increase transitions.

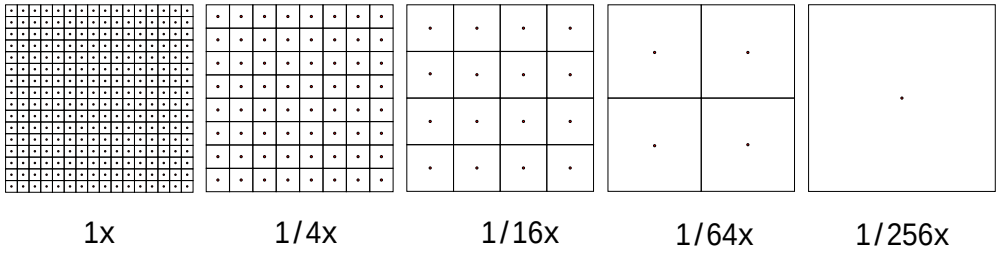


Figure 6.6: The five sampling rates considered in our experiments, from 1x (left) to 1/256x (right).

As images generated with lower sampling rates have fewer high-frequency components and different sampling rate requirements, each transition in the FSM has individual values for $< T_R, D_R >$ and for $< T_I, D_I >$. Apparently, the FSM has 4 Increase and 4 Decrease transitions. However, at 1/256x rate, fragments are sampled just once and the resulting tile contains a single plain color. As there is no spatial frequency in it, the heuristic cannot make decisions based on the coefficient matrix. Our FSM conservatively forces the 1/256x state to always transition back to 1/64x. Consequently, parameter values for 3 Increase and 4 Reduce transitions in the FSM must be set. Extensive experiments have been performed to empirically determine adequate values for these parameters that can be universally applied in mobile graphics applications and reduce GPU activity (samples) while keeping the original image quality. Section 6.3 describes the methodology followed to find such optimal $< T_R, D_R >$, $< T_I, D_I >$ values for each sampling rate.

6.3 Heuristic Parameter Selection

This section describes the empirical methodology to find the best values for DSR parameters such that frames are rendered at the lowest possible average sample rate (ASR) without producing any visible error.

As depicted in Algorithm 5, an exhaustive parameter exploration is performed. For each parameter combination under test all frames are rendered, adjusting the sample rate of each tile according to the output of the heuristic. The 100-frame traces from the 20 applications employed to evaluate the techniques described in Chapters 4 and 5 are used to fit the parameters. During the search, any combination that produces even a single erroneous frame is directly discarded. Otherwise, the achieved ASR across all frames is computed for that combination. Eventually, the parameter combination that produces the lowest ASR is chosen.

Frame errors are computed by comparing the image quality of the produced frames with respect to the frame rendered at baseline sampling rate using the Mean Structural Similarity Index (MSSIM [83]), a widely adopted, perceptually-based quality metric that estimates the visual impact of changes in image luminance and contrast caused by compression distortions. The MSSIM has been shown to outperform other similarity metrics that just measure differences in pixel color, such as PSNR and MSE, in terms of quality [24, 42] as it correlates better with the perception of the human visual system. A frame error occurs whenever the obtained MSSIM is lower than 95, as it is the point at which defects can be discerned by human beings [23].

Algorithm 5 Basic parameter search

Input: Parameter combinations to explore, frames to consider.
Output: Values for the $< T_I, D_I >$ and $< T_R, D_R >$ tuples that produce the lowest ASR.

```
1: for each parameter combination do
2:     for each frame do
3:         for each tile do
4:             DCT = compute_dct(tile, frame)
5:             if MaxC(D_R, DCT) < T_R then
6:                 next = SR[tile, frame] − 1                ▷ Reduce
7:             else if MaxC(D_I, DCT) ≥ T_I then
8:                 next = SR[tile, frame] + 1                ▷ Increase
9:             else
10:                next = SR[tile, frame]                    ▷ Stay
11:            end if
12:            SR[tile, frame + 1] = next
13:        end for
14:        if contains_errors(frame) then
15:            discard parameter combination
16:        end if
17:    end for
18:    compute_ASR(parameter_combination, SR)
19: end for
```

Each parameter combination contains a set of 14 different parameters (four $< T_R, D_R >$ pairs for the Reduce transitions and three $< T_I, D_I >$ pairs for the Increase transitions). Even considering just a few values for each parameter (say n), the sheer amount of combinations to consider (n^{14}) makes an exhaustive exploration unfeasible. We adopt instead a divide and conquer approach in which we first only focus on finding the best parameters for the Increase transitions. Next, those values are used and kept constant in Algorithm 5 to find the best parameters for the Reduce transitions. By splitting the parameter search into two steps, we substantially limit the number of combinations to explore and we can execute an exhaustive search.

Note however that during the first step Algorithm 5 cannot be applied: without values set for the $< T_R, D_R >$ pairs, the procedure lacks a mechanism to dynamically reduce the sample rates and the FSM never reaches the lowest states. Consequently, an alternative sampling rate reduction mechanism for this first step must be provided. Such mechanism should produce tiles at low enough sampling rates that Increase transitions are required to prevent errors due to undersampling. Otherwise (if Increase decisions were never required) the capabilities of the parameter combinations to produce a low ASR while not producing frame errors would not be tested.

A simple preliminary experiment that finds near-optimal sample rates for each of the tiles in all frames is first conducted to build an effective reduction mechanism. Those values will act as references and will stay constant during the exploration of the Increase parameters. The reduction mechanism consists in always choosing the lowest sampling rate between the reference value and

the outcome of the heuristic (which either increases the sampling rate or keeps it the same).

This preliminary experiment first generates the images of all tiles in all frames at all five sampling rates. It then sequentially analyzes tile by tile the five alternatives and selects the lowest one that does not produce visible errors compared with the same tile at baseline sampling. These sample rates are named *Local Minimum*, because image discrepancies are not analyzed at full frame level but just at tile level. As such, they may not be the optimal sample rates (optimal values may be lower when discrepancies are analyzed at frame level) but they are low enough to be used as a reference in our reduction mechanism.

Algorithm 6 shows the procedure to find the best parameters for the Increase transitions (the first step). Akin to Algorithm 5, for each tile it computes the DCT and decides whether or not to increase the current sampling rate according to the $< T_I, D_I >$ parameters under test (Lines 5-9). However, unlike Algorithm 5, it next considers overriding that decision by choosing instead the stored Local Minimum sample rate for the next frame (Line 11) in case that it is lower. As the algorithm can select a sampling rate lower than the Local Minimum, the found parameters gravitate towards the optimal sampling rates.

Algorithm 6 LocMin parameter search

Input: $LocalMinimum$, $< T_R, D_R >$, parameter combinations to explore, frames to consider.
Output: Values for the $< T_I, D_I >$ tuples that produce the lowest ASR.

```
1: for each parameter combination do
2:     for each frame do
3:         for each tile do
4:             DCT = compute_dct(tile, frame)
5:             if MaxC(D_I, DCT) ≥ T_I then
6:                 next = SR[tile, frame] + 1                ▷ Increase
7:             else
8:                 next = SR[tile, frame]                    ▷ Stay
9:             end if
10:            locmin = LocalMinimum[tile, frame + 1]
11:            SR[tile, frame + 1] = min(next, locmin)
12:        end for
13:        if contains_errors(frame) then
14:            discard parameter combination
15:        end if
16:    end for
17:    compute_ASR(parameter_combination, SR)
18: end for
```

6.4 Implementation

This section describes the combinational logic and memory structures required to implement Dynamic Sampling Rate, and how the frequency analysis and sample rate determination are integrated within the Raster Pipeline.

6.4.1 Pipeline Integration

The Dynamic Sample Rate technique uses a FSM (see Figure 6.5) to dynamically determine the sampling rate of each tile based on its current state and its *MaxC*. It requires a new hardware structure called *Sampling Rate Table* (SRT), with one entry per tile, that holds the state of each tile in a frame. Since the FSM has 5 different states, a state can be represented with 3 bits. Consequently, for the frame resolution of 1080x1920 pixels used in our experiments, there are 8100 tiles and the storage overhead of the SRT is 2.96 KB. The SRT has been modelled as a McPAT SRAM to obtain its power and area.

Other than the SRT, Dynamic Sampling Rate requires very minor modifications to the pipeline, as shown in Figure 6.7. Tiles are scheduled and primitives are fetched in the same way as in the baseline because the sampling rate only affects the discretization process. The Rasterizer still produces Quads (square groups of four adjacent fragments), so they can be depth tested, shaded and blended as in the baseline. The main difference is that the screen area covered by each fragment is bigger than a pixel when the sampling rate is lower than 1x. Those fragments are named *Superfragments* and a group of four Superfragments is named *Superquad*. Producing a superfragment at a sampling rate of 1/NxN only requires sampling at the center of a grid of NxN pixels.

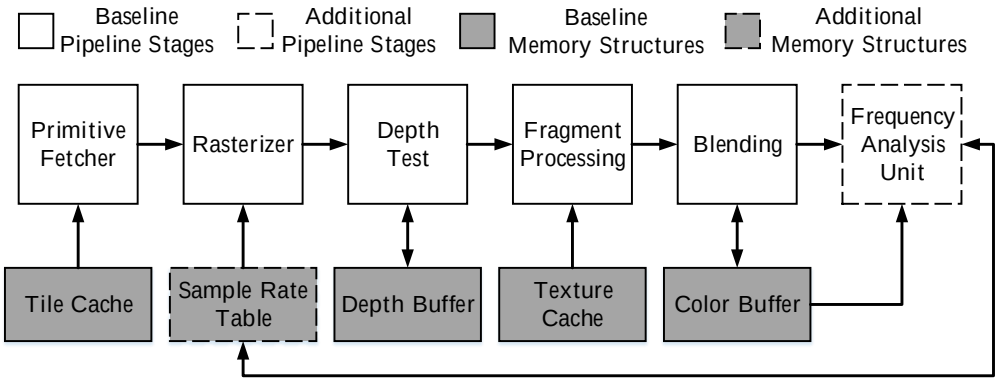


Figure 6.7: Raster Pipeline with DSR.

Whenever a tile starts its processing, its state (hence the associated sampling rate) is fetched from the Sampling Rate Table. The Rasterizer generates Superfragments according to the stored state. The Depth and Color Buffers already have capacity to hold temporary values for the 256 pixels (16x16 pixel tiles) of the baseline resolution. Fragments within a Superfragment share depth and color, so, only one read/write operation in the Depth Buffer is executed when depth testing a Superfragment and only one read/write operation in the Color Buffer is executed when blending

a Superfragment. This results in some entries of the Color Buffer not being initialized after a tile finishes its processing. In the last pipeline stage, the final color value of a Superfragment is upsampled by replicating a color to all pixels belonging to it. Afterwards, the contents of the Color Buffer are transferred to main memory and the DCT computation of the tile starts

6.4.2 Frequency Analysis Unit

The 2D DCT is a separable function [60]. This property allows to transform a NxN *Input* image into the frequency domain by successively applying 1D transforms, first along the rows and then along the columns (or vice-versa). By considering separability, the well-known 2D-DCT formula can be rearranged as shown in Equation 6.1:

$$DCT(p,q) = \alpha(p)\alpha(q) \sum_{m=0}^{N-1} cos\frac{(2m+1)\pi p}{2N} \sum_{n=0}^{N-1} Input_{mn} cos\frac{(2n+1)\pi q}{2N} \quad (6.1)$$

where $0 \leq p, q \leq N-1$ and the scale factors α are defined as:

$$\alpha_p = \alpha_q \begin{cases} \frac{1}{\sqrt{N}} & \text{if p = 0 or q = 0} \\ \sqrt{\frac{2}{N}} & \text{otherwise.} \end{cases} \quad (6.2)$$

The 2D-DCT formula is usually expressed in matrix notation as [68]:

$$DCT = KInputK^T = (K(KInput)^T)^T \quad (6.3)$$

where K is the so-called *Kernel Matrix*, that contains precomputed values for both the scale factors and the cosine functions in the form of:

$$K_{pq} = \begin{cases} \frac{1}{\sqrt{N}} & \text{if p = 0} \\ \sqrt{\frac{2}{N}} cos\frac{(2q+1)\pi p}{2N} & \text{otherwise.} \end{cases} \quad (6.4)$$

DSR's frequency analysis scheme uses the Synopsys's implementation of the 2D DCT transform from their DesignWare library [72]. This module is based on the aforementioned Kernel Matrix precomputation and row-column decomposition. The computation of the 2D DCT shown in Equation 6.3 is divided in two steps, decoupled by an auxiliary buffer (Aux) that holds temporary results. The first step computes the 1D-DCT of the rows ($Aux = (KInput)^T$) and the second step completes the 2D computation ($DCT = (KAux)^T$). In the Synopsys implementation, a single buffer is used for storing both the temporary and final results. This buffer, named *DCT Buffer*, is written by columns and read by rows to emulate the two transpositions.

Figure 6.8 shows a block diagram of the Frequency Analysis Unit and its dataflow: the input data is read from the Color Buffer (1) and is multiplied by the Kernel Matrix (2) using a series of compute units. Each unit computes the 1D-DCT of a row and stores the result in the DCT Buffer (3). Since the tiles in our modeled GPU are composed of 16x16 pixels and the frequency analysis unit contains 4 compute units, each unit sequentially processes 4 rows. Once the 16 rows have been processed, the second pass is performed: the temporary contents of the DCT Buffer (4) are multiplied to the Kernel Matrix and stored back in the DCT Buffer (5), four columns at a time, until the final 2D DCT is computed.

The original Synopsys design operates sequentially in each row and column, as it does not have hardware to compute multiple 1D-DCTs in parallel. This implies significant time overheads to compute the entire 2D-DCT of 16x16 elements. The design has been slightly modified by replicating the compute units. Experimentally, it has been determined that with 4 compute units the frequency analysis and sampling rate determination do not cause stalls in the pipeline and the energy and area overheads are minimal (results in Section 6.5). The Frequency Analysis Unit has been implemented in VHDL and synthesized to obtain its delay and power using the Synopsys Design Compiler, the modules of the DesignWare library and the 32/28nm technology library from Synopsys [73].

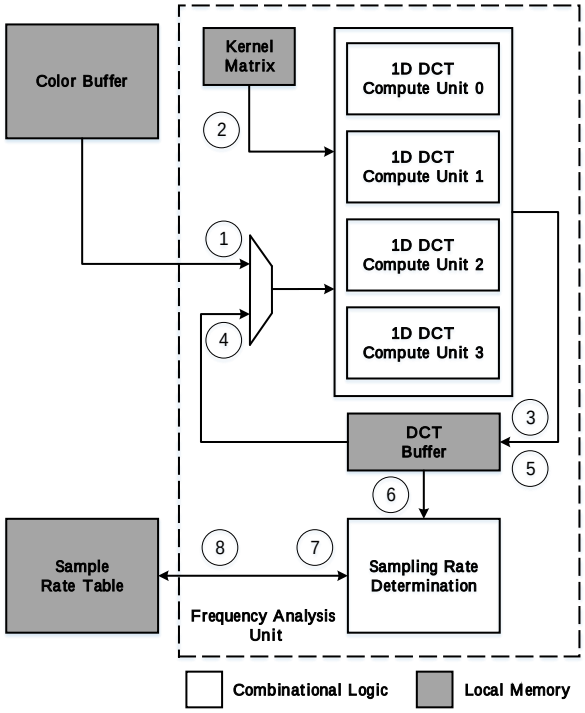


Figure 6.8: Frequency Analysis Unit overview.

Once the DCT computation ends, the hardware (6) estimates the best sampling rate for that tile using the scheme presented in Section 6: it first uses the matrix of coefficients (ignoring the D first diagonals) to compute $MaxC$; then, following the FSM in Figure 6.5, it decides the new tile state (hence a corresponding sampling rate), based on the current state (7) and the comparison between $MaxC$ and the T threshold. Finally, the new tile state is stored in the SRT (8) to indicate

the sampling rate to be used in the following frame.

6.5 Experimental Results

This section presents the main results of Dynamic Sampling Rate over the baseline architecture, described in Table 3.1. Results are reported over a different set of benchmarks than the ones used to fit the heuristic parameters.

Figure 6.9 shows the energy consumption of the whole system (GPU plus memory) with the proposed Dynamic Sampling Rate approach normalized to the baseline architecture. It can be seen that having independent and dynamic sampling rates for each tile achieves an average 40% reduction of energy, with savings up to 67% (for *Dragon Ball*). Figure 6.9 also shows the minor costs of activating DSR: the static and dynamic energy consumption of the Sampling Rate Table, and the logic and temporary memory required to compute the 2D DCT of the tiles (Figure 6.8). All together, they represent less than 2% of the total energy consumption and less than 1% of the area of the baseline GPU.

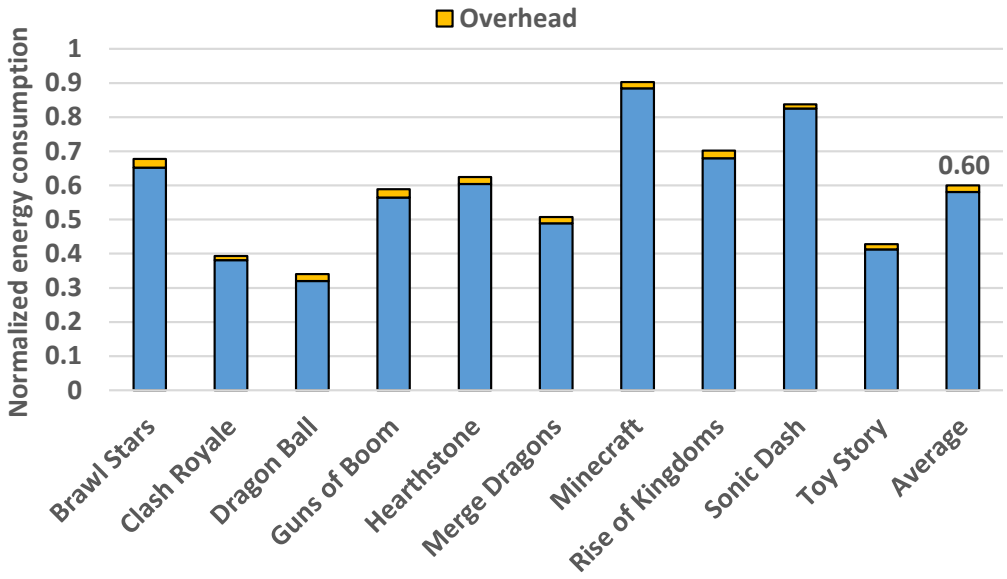


Figure 6.9: Energy consumption of DSR compared to the Baseline GPU.

Figure 6.10 shows the reduction in execution cycles of DSR normalized to the Baseline design and broken down into Geometry and Raster cycles. On average, the presented proposal leads to 1.9x speedup in the Raster Pipeline, with maximums of more than 4x (*Dragon Ball*). This translates into a 36% global execution time reduction, since the Geometry Pipeline contains no modifications with respect to the baseline. Note that DSR does not incur in any execution time penalty, as the frequency analysis of the tiles and their sampling rate determination is completely overlapped with the Raster Pipeline activity.

The benefits in energy consumption and execution time of DSR are provided by sampling most

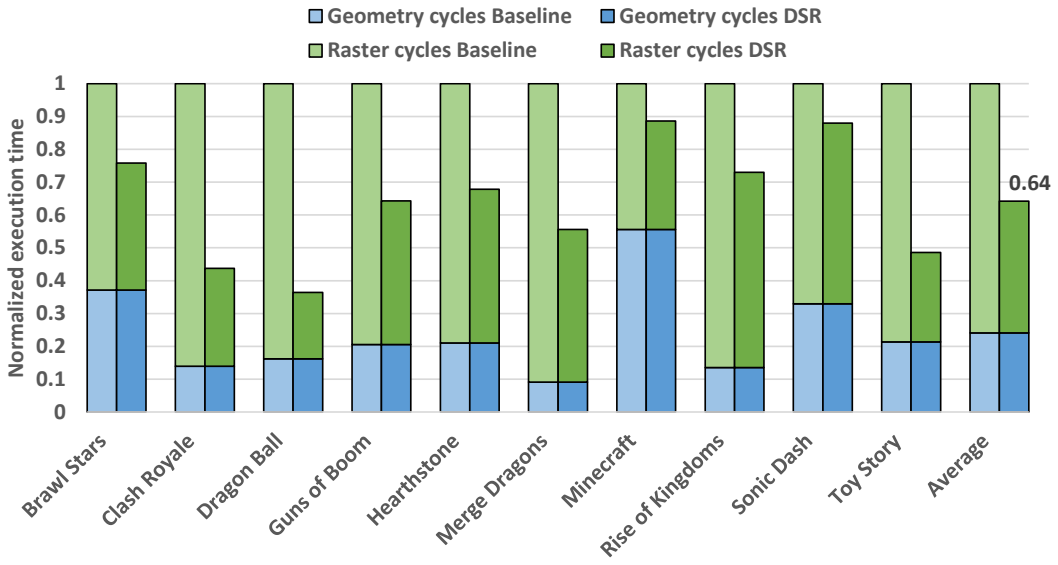


Figure 6.10: Execution time of DSR compared to the Baseline GPU.

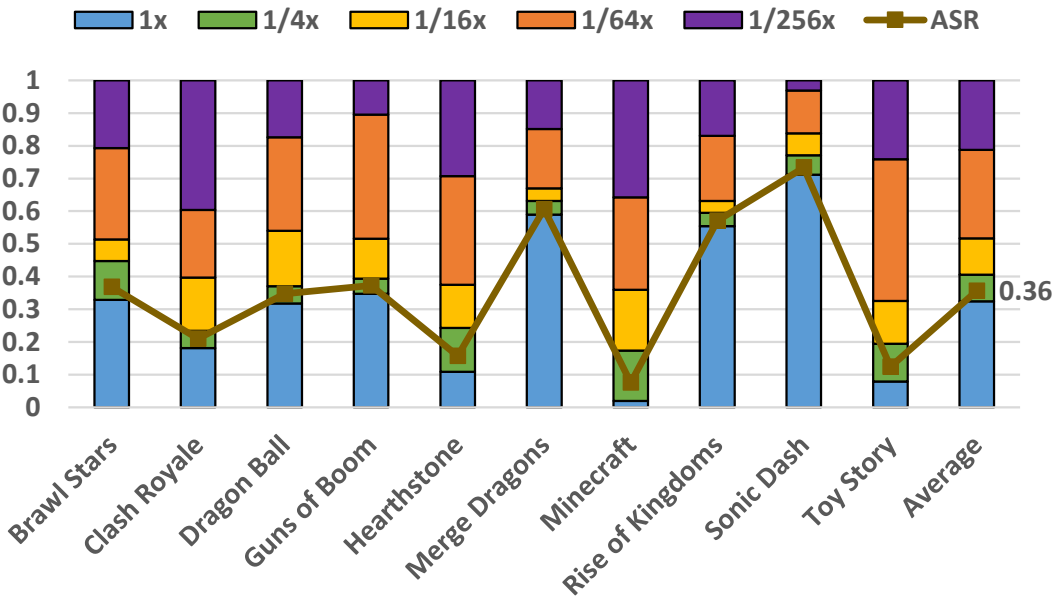


Figure 6.11: Breakdown of sampling rates.

tiles at lower rates, as shown in Figure 6.11. On average, less than half of the tiles are sampled at the baseline rate, while almost 40% of the tiles are processed using the two lowest sampling rates (1/64x and 1/256x). The Average Sample Rate across all benchmarks and frames is thus reduced to 0.36 samples per fragment. This greatly reduces the activity of the Fragment Shaders, as shown in Figure 6.12. DSR reduces the average number of processed fragments by 66% and the number of texture accesses to main memory by 28% when compared to the Baseline. The gap between both numbers is caused by an increase in sparsity: as samples are taken at larger intervals, the likelihood of reusing a texture cache line is smaller than in the Baseline. However, the great

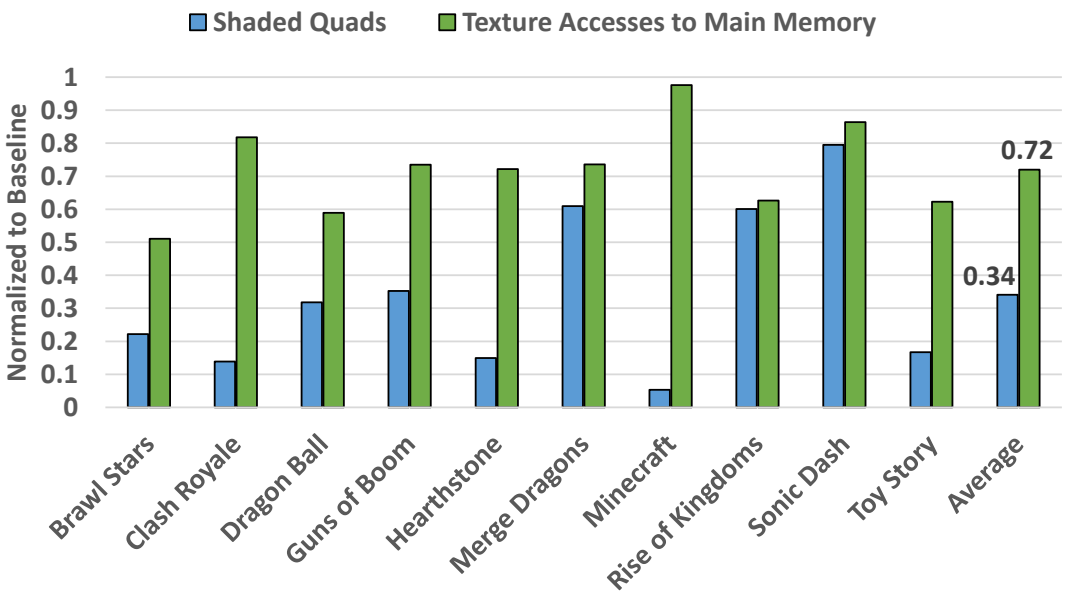
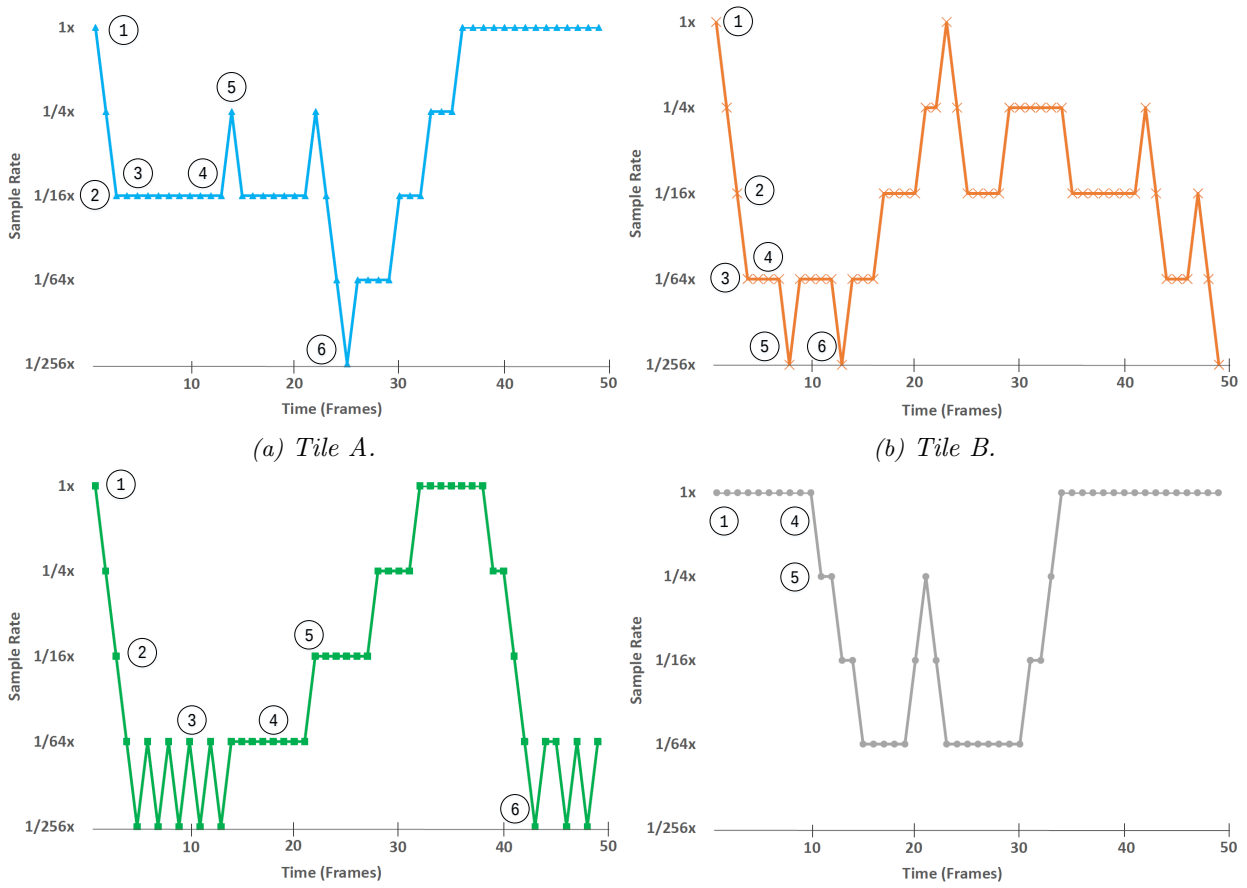


Figure 6.12: Shader activity of Dynamic Sampling Rate compared to the Baseline GPU.

reduction in processed fragments still allows for significant savings in overall texture traffic.

Rendered scenes in real-time applications tend to smoothly vary across consecutive frames. Therefore, the sampling rate requirements of tiles may evolve over time. As DSR analyzes the frequencies of the scene after rendering each tile, it manages to dynamically capture such changes and quickly adjust the sampling rates of all individual tiles accordingly. This process is illustrated in Figure 6.13, which shows the changes in the sampling rate of 4 example tiles of benchmark *Clash Royale* that exhibit different behaviors. DSR starts sampling all the tiles at the maximum rate, 1x (①). Tiles A,B and C can be sampled at a much lower sampling rate, and spend a small transitory period of time continuously reducing their sampling rate (②) until their optimal rate for their current spatial frequency is found (③). Tiles remain in their estimated-optimal sampling rates (④) until the spatial frequencies in them change (e.g., ⑤). Note how every time that a tile is sampled at 1/256x rate (e.g., ⑥), the sampling rate is immediately increased in the following frame, as described in Figure 6.5. It can be observed that the scene is not sampled uniformly neither in space (in a particular frame the sampling rate of the 4 tiles is normally different) nor in time (the sampling rate of a particular tile changes across the frames).

In the performed experiments, DSR has not produced a single error in all the generated frames, i.e., it has not rendered any frame with a MSSIM lower than 95 when compared with the frame rendered at baseline sampling rate. Despite using benchmarks containing swift camera movements and object displacements across the screen, the similarity between consecutive frames allows the reuse of the estimated-best sampling rates without producing any visual artifacts. Albeit a sparse phenomenon, more abrupt alterations may occur in a particular frame, such as in a change of scene. The correctness of DSR cannot be guaranteed in these rare scenarios, as it is based on frame coherency. We have performed an experiment to quantify the effect that DSR has on image quality whenever there is a scene change. To do so, three additional 100-frame traces for the benchmarks



(a) Tile A. *(b) Tile B.*

(c) Tile C. *(d) Tile D.*

Figure 6.13: Evolution of the sampling rate of 4 different tiles over several frames.

listed in Table 6.1 have been generated. Each trace contains two different scene changes, emulated by entering and exiting the pause or settings menu of the application. With the renderization of these applications' frames with DSR active, it has been observed that only the frame rendered immediately after each of the six scene changes is erroneous. Subsequent frames are indistinguishable from frames rendered at baseline sampling rate. It is well documented that the human eye requires some time to construe visual information: if scenes are presented as rapid sequence of pictures, at least 67ms are needed to identify large objects and recreate the essence of the scene [26] while at least 40ms are required to identify an object in motion instead of two simultaneous objects [34]. These times are greater than what a single frame lasts in 30 frames per second, the frame rate which is considered to be the minimum acceptable [37, 16], so it can be concluded that these potential errors affect only a single frame and will not be perceived by the user.

Table 6.1: Additional benchmarks for the image quality experiment.

Benchmark	Genre
Alto's Odyssey	Endless Runner
PlayerUnknown's Battlegrounds	Battle Royale
Homescapes	Puzzle

6.6 Conclusions

This chapter has proposed Dynamic Sampling Rate (DSR), a novel microarchitectural technique to reduce shader executions by determining the lowest sampling rate for each tile in a frame that does not reduce the overall quality of the rendered images. DSR analyzes the frequency components of a tile once it has been processed and decides the rate in which the tile's triangles will be sampled in the following frame. The sampling rate prediction leverages the frame-to-frame coherence inherent in animated graphics applications, which results in a high likelihood that the frequency components of a tile are maintained across consecutive frames.

For a set of unmodified commercial Android applications, DSR reduces the fragment-level redundancy by 66% on average with minimal hardware overhead, leading to an average speedup of 1.68x and energy savings of 40%.

7 Conclusions

This chapter presents the main contributions of this thesis and suggests some potential open-research areas for future work.

7.1 Conclusions

This thesis has focused on improving the energy efficiency of mobile devices at running graphics applications, one of the most popular types of use cases nowadays, which, unfortunately, requires an unsustainable power supply for a battery-operated GPU.

An initial analysis was performed to pinpoint which stages of the graphics pipeline in a Tile-Based Rendering architecture consume the most energy, revealing that fragment processing and especially, its accesses to off-chip main memory is the main contributor. Three techniques have been proposed to reduce the activity at the fragment processors by leveraging frame coherence, a fundamental property of graphics applications that results in consecutive frames producing very similar outputs.

In first place, frame coherence was employed to reduce the number of redundant tiles. It was quantified in a variety of popular commercial Android applications that an average of 62% of the tiles have exactly the same colors across consecutive frames. Given that Tile-Based Rendering decouples geometry from fragment processing, all the inputs required to process a tile are known before rendering it. **Rendering Elimination** has been presented as a mechanism to discard the processing of a tile and reuse the result produced in the previous frame by means of comparing the signatures of the entire set of the tile's inputs between frames. Frame coherence ensures that a significant portion of tiles will maintain the same inputs between frames. Rendering Elimination's implementation introduces very little overheads in energy, area and time, as it is designed to

incrementally compute tile signatures with a series of small lookup tables operating in parallel with geometry processing. Results show that Rendering Elimination avoids the execution of 50% of the tiles, yielding energy savings of 37% and an execution time reduction of 33%.

In second place, frame coherence was utilized to estimate visibility early in the pipeline. Visibility resolution is performed in current GPUs at the very end of the pipeline, which leads to substantial energy waste in overshading, the processing of primitives which eventually do not contribute to the final colors of the image. **Early Visibility Resolution** was presented as a mechanism to gather the resolved visibility information of a frame in order to guide the execution of the following one, as frame coherence ensures that the visibility of most primitives is maintained between frames. The farthest visible point of a tile is computed and stored, and used as a visibility threshold: if the nearest vertex of a primitive binned into a tile is farther than the farthest visible point for that tile in the previous frame, it is considered that the primitive will be occluded in the current frame as well. This estimation is used to remove ineffectual computations at two different granularities: On the one hand, primitives predicted to be occluded are scheduled to be processed after primitives predicted to be visible. Therefore, the effectiveness of the Early Depth Test increases and fragment-level redundancy is reduced. On the other hand, primitives predicted to be occluded are not included in Rendering Elimination's tile signature. Therefore, the effectiveness of Rendering Elimination increases and tile-level redundancy is reduced. Early Visibility Resolution's implementation consists in three small on-chip buffers to compute, store and query visibility information, introducing very little overheads in area and energy consumption. Results show that with Early Visibility Resolution the Early Depth Test rejects 20% more fragments and Rendering Elimination discards 5% more tiles, yielding energy savings of 43% and an execution time reduction of 39%.

In third place, frame coherence was leveraged to reduce the number of fragment shader executions in tiles with low spatial frequencies. Current GPUs rasterize triangles by sampling them once per pixel, but an analysis revealed that almost half of the screen can be processed at a lower sampling rate without affecting image quality. **Dynamic Sampling Rate** was presented as a mechanism to analyze the spatial frequencies of tiles once they have been rendered and decide, based on a heuristic, the best sampling rate for them, which is applied in the following frame. Frame coherence ensures that most tiles maintain the same spatial frequency between frames. Dynamic Sampling Rate's implementation consists in a small hardware unit that evaluates the spatial variations of a rendered Color Buffer in parallel with the processing of the subsequent tile, introducing very little overheads in area and energy consumption. Results show that Dynamic Sampling Rate reduces the fragment-level redundancy by 66%, yielding energy savings of 40% and an execution time reduction of 36%.

7.2 Open-Research Areas

The three techniques presented during this thesis can be further developed in new directions:

- Rendering Elimination is able to discard most of the tiles that produce the same results across consecutive frames, but not all of them. The restriction that all the input set must be exactly the same is, seemingly, too restrictive. With Early Visibility Reduction, the number of

redundant tiles discarded was increased by removing occluded primitives from the considered input set, but still a significant portion remain undetected. There is a wide variety of factors that could contribute to different vertex attributes leading to the same colors for a tile, from big and homogeneous textures to precision round-off errors. The impact of all these reasons could be studied in order to devise an improved Rendering Elimination that considered them.

- Early Visibility Resolution was designed with several approximations to ensure a simple implementation: visibility is estimated by comparing only the nearest vertex of a primitive to only the farthest visible depth of the tile in the previous frame. While the processing of a large amount of occluded fragments is avoided this way, applications still contain a non-negligible amount of overshading. Finer granularity information could be used to enhance the visibility prediction. For instance, multiple depths could be stored per tile and the depth of a primitive could be compared to one or more values to predict its visibility. These additional depths could represent a wide variety of concepts, such as the farthest visible point in sub-regions of a tile, several representative depths between the closest and the farthest values in the tile, or a hierarchy of depths.

- Dynamic Sampling Rate considered sampling once per fragment to be the highest possible sampling rate. Some GPUs allow higher sampling rates, a technique known as Supersampling Antialiasing, which generates images of improved quality by means of sampling triangles multiple times per pixel. Supersampling Antialiasing is normally not applied in graphics applications because increasing the sampling rate for the entire scene has a major performance impact. Dynamic Sampling Rate could be employed to selectively apply Supersampling in the regions of the screen that really require it. Additionally, the energy savings gained from undersampling could be estimated so that Supersampling is applied in a number of tiles so that the Baseline energy consumption is not exceeded. Current GPUs employ a technique to reduce aliasing artifacts without increasing shading costs known as Multisampling Antialiasing (MSAA) [2]. MSAA decouples coverage from shading by sampling only depth at several points in the pixel and sharing the shader results of a single shader execution sampled at the center of the pixel with all the coverage samples that pass the Depth Test. Dynamic Sampling Rate can be combined as is with MSAA, but MSAA could also be extended with information received from the frequency analysis in order to dynamically adjust the number of coverage samples in each tile and both make a better use of resources and produce higher-quality images.

This thesis has shown that frame coherence can be leveraged to reduce the workload of the fragment shaders by means of eliminating redundant activity. There are a plethora of additional ways to use knowledge gained in a previous frame to influence the execution of the following one to improve energy efficiency. For instance, it is expected that the memory access pattern of the entire scene will be very similar between frames. Such memory accesses include, in the Geometry Pipeline, the vertices that are fetched and the primitives that are stored and, in the Raster Pipeline, the primitives and textures that are fetched. The access pattern in any of these pipeline stages could be inspected and noted in a frame in order to schedule the subsequent frame's workload in a more cache-friendly manner to reduce main memory accesses and, consequently, energy consumption.

The proposals of this thesis have been developed on a GPU architecture compliant with the OpenGL ES 2.0 specification. Newer, backwards-compatible versions of the API have been released

by the Khronos Group, which include powerful features to enhance the visual quality of graphics applications. Several of these features are ideal candidates for research in energy efficiency, as they imply a significant increase of the workload:

- Multiple Render Targets. Modern GPUs allow the scene to be rendered differently at several intermediate buffers that can later be used as inputs to the fragment shaders. A common use of this feature is deferred shading, where a geometry pass stores attributes such as normals, diffuse colors and depths in different buffers and then the fragment processing uses that information to compute the direct and indirect lightning effects at each pixel. Multiple Render Targets allow for more realistic effects at the cost of more memory accesses and complex shaders that are able to produce multiple results and later consume them. A potential line of research could be to analyze these memory accesses and localize areas to reduce them.

- Tesselation. Geometric models are designed with a static polygon count, which is a trade-off between the desired level of detail and the cost of transferring the model to the GPU. Modern GPUs allow the hardware to generate primitives by subdividing triangles into smaller triangles, dynamically regulating the level of detail in regions or objects specified by the programmer. While tesselation heavily alleviates vertex fetching and processing, it increases the number of generated primitives which, in turn, increases the number of writes and posterior reads in the parameter buffer. Tile-Based Rendering is currently employed as a low-power design because the overhead of using the Parameter Buffer to decouple the Geometry and Raster pipelines is heavily offset by the benefits obtained by processing raster workload one tile at a time. However, if the reads and writes to the Parameter Buffer were to dramatically increase, architectural solutions would need to be found in order to keep the GPU energy-efficient.

- Compute Shaders. Older generations of GPUs had specific-purpose cores with a relatively low complexity, as the graphics pipeline greatly limits the computational capabilities of vertex and fragment shader programs with restrictions such as the specific timing in which they are executed or the data types they must output. On the other hand, cores in current GPUs can be used for general purpose computing, where a task sent to the GPU by the application is divided into small chunks which are executed in parallel and independently of the graphics pipeline. As flexibility is added to the programming model, the design of the GPU cores becomes more intricate and their microarchitecutre must undergo substantial changes that may affect the overall balance of the graphics pipeline. In particular, and unlike vertex and fragment shaders, the results of compute shaders are communicated to other stages of the pipeline by writing them into memory, which may greatly increase memory pressure and energy consumption if not properly considered.

Bibliography

[1] Nasir Ahmed, T. Natarajan, and Kamisetty Rao. "Discrete cosine transform". In: *IEEE transactions on Computers* 100.1 (1974), pp. 90–93.

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

[3] Tomas Akenine-Möller and Jacob Ström. "Graphics for the masses: a hardware rasterization architecture for mobile phones". In: *Transactions on Graphics*. Vol. 22. 3. ACM. 2003, pp. 801–808.

[4] Magnus Andersson, Jon Hasselgren, and Tomas Akenine-Möller. "Masked depth culling for graphics hardware". In: *Transactions on Graphics* 34.6 (2015), pp. 1–9.

[5] Android. *Android Studio*. 2019. URL: https://developer.android.com/studio/index.html.

[6] Antutu. *Antutu Benchmark*. 2018. URL: https://antutu.com.

[7] Apple. *Apple Worldwide Developers Conference Keynote*. 2018. URL: https://developer.apple.com/videos/play/wwdc2018/101/.

[8] Jose-Maria Arnau, Joan-Manuel Parcerisa, and Polychronis Xekalakis. "Eliminating redundant fragment shader executions on a mobile gpu via hardware memoization". In: *ACM/IEEE 41st Intl. Symp. on Computer Architecture*. IEEE. 2014, pp. 529–540.

[9] Jose-Maria Arnau, Joan-Manuel Parcerisa, and Polychronis Xekalakis. "Parallel frame rendering: Trading responsiveness for energy on a mobile gpu". In: *Proceedings of the 22nd international conference on Parallel architectures and compilation techniques*. IEEE. 2013, pp. 83–92.

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

[11] Jiří Bittner, Michael Wimmer, Harald Piringer, and Werner Purgathofer. "Coherent hierarchical culling: Hardware occlusion queries made useful". In: *Computer Graphics Forum*. Vol. 23. 3. Wiley Online Library. 2004, pp. 615–624.

[12] Aaron Carroll, Gernot Heiser, et al. "An Analysis of Power Consumption in a Smartphone." In: *USENIX annual technical conference*. Vol. 14. Boston, MA. 2010, pp. 21–21.

[13] Ge Chen, Pedro V Sander, Diego Nehab, Lei Yang, and Liang Hu. "Depth-presorted triangle lists". In: *ACM Transactions on Graphics* 31.6 (2012), 160:1–160:9.

[14] Intel Corporation. *Early Z Rejection*. 2012. URL: https://software.intel.com/en-us/articles/early-z-rejection-sample.

[15] Enrique De Lucas, Pedro Marcuello, Joan-Manuel Parcerisa, and Antonio Gonzalez. "Visibility rendering order: Improving energy efficiency on mobile gpus through frame coherence". In: *IEEE Transactions on Parallel and Distributed Systems* 30.2 (2018), pp. 473–485.

[16] Kurt Debattista, Keith Bugeja, Sandro Spina, Thomas Bashford-Rogers, and Vedad Hulusic. "Frame rate vs resolution: A subjective evaluation of spatiotemporal perceived quality under varying computational budgets". In: *Computer Graphics Forum*. Vol. 37. 1. Wiley Online Library. 2018, pp. 363–374.

[17] Xavier Décoret. "N-Buffers for efficient depth map query". In: *Computer Graphics Forum* 24.3 (2005), pp. 393–400.

[18] Deloitte. *Global mobile consumer survey*. 2018. URL: https://www2.deloitte.com/content/dam/Deloitte/ch/Documents/technology-media-telecommunications/ch-deloitte-en-global-mobile-consumer-survey-2018.pdf.

[19] Deloitte. *Global mobile consumer trends, 2nd edition*. 2018. URL: https://www2.deloitte.com/content/dam/Deloitte/us/Documents/technology-media-telecommunications/us-global-mobile-consumer-survey-second-edition.pdf.

[20] ARM Developer. *ARM Mali GPU OpenGL ES Application Optimization Guide*. 2019. URL: https://developer.arm.com/docs/dui0555/latest/api-level-optimizations/minimize-draw-calls/about-minimizing-draw-calls.

[21] Flaunt Digital. *The evolution of mobile phones: 1973 to 2019*. 2018. URL: https://flauntdigital.com/blog/evolution-mobile-phones/.

[22] Cass Everitt. *Interactive order-independent transparency*. 2001. URL: https://www.nvidia.com/en-us/drivers/Interactive-Order-Transparency.

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

[24] Xinbo Gao, Wen Lu, Dacheng Tao, and Xuelong Li. "Image quality assessment based on multiscale geometric analysis". In: *IEEE Transactions on Image Processing* 18.7 (2009), pp. 1409–1423.

[25] GlobalWebIndex. *Which smartphone features really matter to consumers?* 2019. URL: https://blog.globalwebindex.com/chart-of-the-week/smartphone-features-consumers/.

[26] E. Bruce Goldstein and James Brockmole. *Sensation and perception*. Cengage Learning, 2016.

[27] *GAPID*. 2019. URL: https://developers.google.com/vr/develop/unity/gapid.

[28] Google. *How to ptimize an application using Gapid*. 2019. URL: https://gapid.dev/tutorials/optimize.

[29] Naga K Govindaraju, Michael Henson, Ming C Lin, and Dinesh 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*. ACM. 2005, pp. 49–56.

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

[31] Markus Hadwiger. *GPU Texturing*. 2015. URL: https://faculty.kaust.edu.sa/sites/markushadwiger/Documents/CS380_spring2015_lecture_12.pdf.

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

[33] Yong He, Yan Gu, and Kayvon Fatahalian. "Extending the graphics pipeline with adaptive, multi-rate shading". In: *Transactions on Graphics* 33.4 (2014), pp. 1–12.

[34] Michael H Herzog, Thomas Kammer, and Frank Scharnowski. "Time slices: what is the duration of a percept?" In: *PLoS biology* 14.4 (2016), e1002433:1–e1002433:12.

[35] Innospective. *The Smartphone Revolution: Why the App Store Was More Important Than the iPhone*. 2018. URL: https://flauntdigital.com/blog/evolution-mobile-phones/.

[36] Apple Insider. *How AMD and NVIDIA lost the mobile GPU business to Apple*. 2015. URL: https://appleinsider.com/articles/15/01/23/how-amd-and-nvidia-lost-the-mobile-gpu-chip-business-to-apple-with-help-from-samsung-and-google-.

[37] Benjamin F Janzen and Robert J Teather. "Is 60 FPS better than 30?: the impact of frame rate and latency on moving target selection". In: *Proceedings of the extended abstracts of the 32nd annual ACM conference on Human factors in computing systems*. ACM. 2014, pp. 1477–1482.

[38] Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. "McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures". In: *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*. 2009, pp. 469–480.

[39] Gábor Liktor and Carsten Dachsbacher. "Decoupled deferred shading for hardware rasterization". In: *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*. 2012, pp. 143–150.

[40] Arm Limited. *How low can you go? Building low-power, low-bandwidth ARM Mali GPUs*. 2013. URL: https://community.arm.com/developer/tools-software/graphics/b/blog/posts/how-low-can-you-go-building-low-power-low-bandwidth-arm-mali-gpus.

[41] Arm Limited. *Transaction Elimination*. 2014. URL: https://developer.arm.com/technologies/graphics-technologies/transaction-elimination.

[42] Qi Ma, Liming Zhang, and Bin Wang. "New strategy for image and video quality assessment". In: *Journal of Electronic Imaging* 19.1 (2010), 011019:1–011019:14.

[43] Ian Mallett and Cem Yuksel. "Deferred adaptive compute shading". In: *Proceedings of the Conference on High-Performance Graphics*. 2018, pp. 1–4.

[44] Mobile Marketing. *Tapping into the boom in hyper-casual games*. 2019. URL: https://mobilemarketingmagazine.com/tapping-into-the-boom-in-hyper-casual-games.

[45] James Massey. "Shift-register synthesis and BCH decoding". In: *Transactions on Information Theory* 15.1 (1969), pp. 122–127.

[46] Jeffrey J McConnell. *Computer graphics: theory into practice*. Jones & Bartlett Learning, 2005. Chap. 5.2.1.

[47] Trapper Mcferron and Adam Lake. *Checkerboard Rendering for Real-Time Upscaling on Intel Integrated Graphics*. 2018. URL: https://software.intel.com/en-us/articles/checkerboard-rendering-for-real-time-upscaling-on-intel-integrated-graphics.

[48] Micron. *TN-41-01: Calculating Memory System Power for DDR3*. 2007. URL: https://www.micron.com/~/media/Documents/Products/Technical%5C%20Note/DRAM/TN41_01DDR3_Power.pdf.

[49] Steven Molnar, Michael Cox, David Ellsworth, and Henry Fuchs. "A sorting classification of parallel rendering". In: *IEEE computer graphics and applications* 14.4 (1994), pp. 23–32.

[50] Jae-Ho Nah, Yeongkyu Lim, Sunho Ki, and Chulho Shin. "Z^2 traversal order: An interleaving approach for VR stereo rendering on tile-based GPUs". In: *Computational Visual Media* 3.4 (2017), pp. 349–357.

[51] Mozilla Developer Network. *Frame Rate and responsiveness*. 2019. URL: https://developer.mozilla.org/en-US/docs/Tools/Performance/Frame_rate.

[52] Notesmatic. *Factors that affect demand for smartphones and tablets in the global market*. 2019. URL: https://notesmatic.com/2018/02/factors-affecting-demand-for-smartphones-and-tablets/.

[53] Harry Nyquist. "Certain topics in telegraph transmission theory". In: *Transactions of the American Institute of Electrical Engineers* 47.2 (1928), pp. 617–644.

[54] Jorn Nystad et al. "Adaptive scalable texture compression". In: *Proc. of the Fourth ACM SIGGRAPH/Eurographics Conf. on High-Performance Graphics*. 2012, pp. 105–114.

[55] PayPal. *Digital media consumer study*. 2017. URL: https://www.paypalobjects.com/digitalassets/c/website/marketing/global/shared/global/media-resources/documents/paypal-digital-media-consumer-study-pt2.pdf.

[56] William Wesley Peterson and Daniel T Brown. "Cyclic codes for error detection". In: *Proceedings of the IRE* 49.1 (1961), pp. 228–235.

[57] Juan Pineda. "A parallel algorithm for polygon rasterization". In: *Proceedings of the 15th annual conference on Computer graphics and interactive techniques*. 1988, pp. 17–20.

[58] Imagination Technologies Group plc. *A look at the PowerVR Graphics Architecture: Deferred Rendering*. 2016. URL: https://www.imgtec.com/blog/the-dr-in-tbdr-deferred-rendering-in-rogue/.

[59] Qualcomm. *Trepn Power Profiler*. 2015. URL: https://developer.qualcomm.com/software/trepn-power-profiler.

[60] Kamisetty Ramamohan Rao and Ping Yip. *Discrete cosine transform: algorithms, advantages, applications*. Academic Press Professional, 2014.

[61] Philipp A Rauschnabel, Alexander Rossmann, and M Claudia tom Dieck. "An adoption framework for mobile augmented reality games: The case of Pokémon Go". In: *Computers in Human Behavior* 76 (2017), pp. 276–286.

[62] MIT Technology Review. *Are Smart Phones Spreading Faster than Any Technology in Human History?* 2012. URL: https://www.technologyreview.com/s/427787/are-smart-phones-spreading-faster-than-any-technology-in-human-history/.

[63] Paul Rosenfeld, Elliott Cooper-Balis, and Bruce Jacob. "DRAMSim2: A cycle accurate memory system simulator". In: *IEEE Computer Architecture Letters* 10.1 (2011), pp. 16–19.

[64] Takafumi Saito and Tokiichiro Takahashi. "Comprehensible rendering of 3-D shapes". In: *Proceedings of the 17th annual conference on Computer graphics and interactive techniques*. 1990, pp. 197–206.

[65] Rahul Sathe and Tomas Akenine-Möller. "Pixel Merge Unit". In: *Eurographics (Short Papers)*. 2015, pp. 53–56.

[66] Simon Scheckel and Andreas Kolb. "Min-max mipmaps for efficient 2D occlusion culling". In: *Conference on Computer Graphics, Visualization and Computer Vision*. 2016, pp. 13–16.

[67] Dean Sekulic. *Efficient Occlusion Culling*. GPU Gems, 2005. Chap. 29.

[68] Tero Sihvo and Jarkko Niittylahti. "Row-column decomposition based 2D transform optimization on subword parallel processors". In: *International Symposium on Signals, Circuits and Systems, 2005. ISSCS 2005.* Vol. 1. IEEE. 2005, pp. 99–102.

[69] Michael Stengel, Steve Grogorick, Martin Eisemann, and Marcus Magnor. "Adaptive image-space sampling for gaze-contingent real-time rendering". In: *Computer Graphics Forum*. Vol. 35. 4. Wiley Online Library. 2016, pp. 129–139.

[70] Yan Sun and Min Sik Kim. "High Performance Table-based algorithm for Pipelined CRC calculation". In: *Journal of Communications* 8.2 (2013), pp. 128–135.

[71] Ivan E Sutherland, Robert F Sproull, and Robert A Schumacker. "A characterization of ten hidden-surface algorithms". In: *ACM Computing Surveys* 6.1 (1974), pp. 1–55.

[72] Synopsys. *DesignWare 2D DCT*. 2019. URL: https://www.synopsys.com/dw/ipdir.php?c=DW_dct_2d.

[73] Synopsys. *Synopsys 32/28nm Generic Library*. 2019. URL: https://www.synopsys.com.

[74] Techcrunch. *Videogame revenue tops 43 billion in 2018*. 2019. URL: https://techcrunch.com/2019/01/22/video-game-revenue-tops-43-billion-in-2018-an-18-jump-from-2017/.

[75] Techjury. *14 Mobile Gaming Statistics, 2020 – Insights Into $2.2B Gamers Market*. 2019. URL: https://techjury.net/stats-about/mobile-gaming/.

[76] Techjury. *How many people play mobile games around the world in 2020*. 2019. URL: https://techjury.net/stats-about/mobile-gaming-demographics.

[77] Government Technology. *How Smartphones Revolutionized Society in Less than a Decade*. 2014. URL: https://www.govtech.com/products/How-Smartphones-Revolutionized-Society-in-Less-than-a-Decade.html.

[78] TheTool. *Infographic: The Evolution of The App Stores*. 2017. URL: https://thetool.io/2017/evolution-app-stores-infographic.

[79] Karthik Vaidyanathan et al. "Coarse pixel shading". In: *Proceedings of High Performance Graphics*. Eurographics Association. 2014, pp. 9–18.

[80] VMWare. *Gallium3D*. 2011. URL: https://www.freedesktop.%20org/wiki/Software/gallium.

[81] VMWare. *Tungsten Graphics Shader Infrastructure*. 2012. URL: https://gallium.readthedocs.io/en/latest/tgsi.html.

[82] G Wagner and William Maltz. "Too hot to hold: Determining the cooling limits for handheld devices". In: *Advancements in Thermal Management* 2013 (2013).

[83] Zhou Wang, Alan C Bovik, Hamid R Sheikh, and Eero P Simoncelli. "Image quality assessment: from error visibility to structural similarity". In: *IEEE transactions on image processing* 13.4 (2004), pp. 600–612.

[84] Christoph Weber and Marc Stamminger. "Topological triangle sorting for predefined camera paths". In: *Proceedings of the Conference on Vision, Modeling and Visualization*. Eurographics Association. 2016, pp. 153–160.