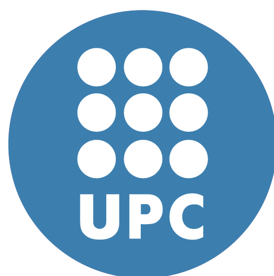
Reducing Redundancy of Real Time Computer Graphics in Mobile Systems

Enrique de Lucas



Doctor of Philosophy

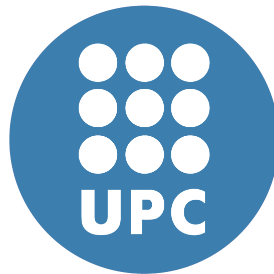
UNIVERSITAT POLITÈCNICA DE CATALUNYA

Department of Computer Architecture

Barcelona, 2018

Reducing Redundancy of Real Time Computer Graphics in Mobile Systems

Enrique de Lucas



ADVISERS
Dr. Joan-Manuel Parcerisa
Dr. Pedro Marcuello

ARCO Research Group
Department of Computer Architecture
UNIVERSITAT POLITÈCNICA DE CATALUNYA
Barcelona
Spain

Thesis submitted for the degree of Doctor of Philosophy at the Universitat Politècnica de Catalunya

‘Gutta cavat lapidem,
[non vi, sed saepe cadendo]’

–Publius Ovidius Naso,
in Epistulae ex Ponto IV, 10, 5.
Expanded in the Middle Ages.

Keywords

Collision Detection, GPU, Android, Rasterization, Mobile GPU, Rendering, Image Based Collision Detection, Graphics Rendering Hardware, Object Interference Detection, Rasterizing Graphics Hardware, Overdraw, Overshading, Rendering Order, Front-to-back, GPU microarchitecture, Visibility, Hidden Surface Removal, Deferred Rendering, Deferred Shading, TBR, TBDR, Temporal Coherence, Frame Coherence, Topological Order, Topological Sort, Energy-efficiency

Abstract

At each new generation the growing computing power of mobile devices has promoted the adoption of more powerful support for graphics and real-time physics simulations with increasing precision and realism. Given the battery-operated and handheld nature of these devices they must use as little energy as possible, because it is crucial to enlarge battery life and to keep a comfortable surface touch temperature. Hence, software and hardware improvements are crucial to deliver a low-power yet rich user experience that satisfies the user demands on the functionality of mobile devices.

The goal of this thesis is to propose novel and effective techniques to eliminate redundant computations that waste energy and are performed in real-time computer graphics applications, with special focus on mobile GPU micro-architecture. Improving the energy-efficiency of CPU/GPU systems is not only key to enlarge their battery life, but also allows to increase their performance because, to avoid overheating above thermal limits, SoCs tend to be throttled when the load is high for a large period of time. Prior studies pointed out that the CPU and especially the GPU are the principal energy consumers in the graphics subsystem, being the off-chip main memory accesses and the processors inside the GPU the primary energy consumers of the graphics subsystem.

In the first place, we focus on reducing redundant fragment processing computations by means of improving the culling of hidden surfaces. During real-time graphics rendering, objects are processed by the GPU in the order they are submitted by the CPU, and occluded surfaces are often processed even though they will end up not being part of the final image. When the GPU realizes that an object or part of it is not going to be visible, all activity required to compute its color and store it has already been performed. We propose a novel architectural technique for mobile GPUs, Visibility Rendering Order (VRO), which reorders objects front-to-back entirely in hardware to maximize the culling effectiveness of the GPU and minimize overshading, hence reducing execution time and energy consumption. VRO exploits the fact that the objects in graphics animated applications tend to keep its relative depth order across consecutive frames (temporal coherence) to provide the feeling of smooth transition. VRO keeps visibility information of a frame, and uses it to reorder the objects of the following frame. Since depth-order relationships among objects are already tested in the GPU, VRO incurs minimal energy overheads. It just requires adding a small hardware to capture the visibility information and use it later to guide the rendering of the following frame. Moreover, VRO works in parallel with the graphics pipeline, so negligible performance overheads are incurred. We illustrate the benefits of VRO using various unmodified commercial 3D applications for which VRO achieves 27% speed-up and 14.8% energy reduction on average.

In the second place, we focus on avoiding redundant computations related to CPU Collision Detection. Graphics animation applications such as 3D games represent a large percentage of

downloaded applications for mobile devices and the trend is towards more complex and realistic scenes with accurate 3D physics simulations. Collision detection (CD) is one of the most important algorithms in any physics kernel since it identifies the contact points between the objects of a scene, and determines when they collide. However, real-time highly accurate CD is very expensive in terms of energy consumption. We propose Render Based Collision Detection (RBCD), a novel energy-efficient high-fidelity CD scheme that leverages some intermediate results of the rendering pipeline to perform CD, so that redundant tasks are done just once. Comparing RBCD with a conventional CD completely executed in the CPU, we show that its execution time is reduced by almost three orders of magnitude (600x speedup), because most of the CD task of our model comes for free by reusing the image rendering intermediate results. Although not necessarily, such a dramatic time improvement may result in better frames per second if physics simulation stays in the critical path. However, the most important advantage of our technique is the enormous energy savings that result from eliminating a long and costly CPU computation and converting it into a few simple operations executed by a specialized hardware within the GPU. Our results show that the energy consumed by CD is reduced on average by a factor of 448x (i.e., by 99.8%). These dramatic benefits are accompanied by a higher fidelity CD analysis (i.e., with finer granularity), which improves the quality and realism of the application.

Acknowledgements

First of all I want to thank my advisers, Joan-Manuel Parcerisa and Pedro Marcuello, who have taught me almost everything I know about Computer Architecture. They always offered me valuable guidance and support as well as they were involved in my day-to-day work. I am truly convinced I could not have had better advisers. I am also very grateful to Prof. Antonio González, who offered me the opportunity to start in this research area, first at Intel Barcelona and then at UPC[1]. Thanks for your sagacious feedback and your wise advise.

I wish to thank the members of ARCO I met these years. Especially to José María Arnau, whose outstanding and pioneering work encouraged me to work in the field of low-power GPUs. I am grateful for all his support and help and for teaching me what an international conference is about. Thanks to Martí -the one- for continuing the research line in ARCO group. Thank you Gem and Emilio for your feedback and friendship. I also wish to thank the "The D6ers": Albert, Franyell, Hamid, Josue, Marc, Martí -the other one- and Reza. I wish you the best of luck. I am lucky to have met all my friends from Campus Nord. Enric, Gemma, Javi, Manu, Marc, Niko and Oscar, who welcomed me to Barcelona and shared with me seminars, meetings, coffees, dinners and parties. However, they did not warn me about where I got when I started the doctorate.

I am really blessed to have so many great friends like Alex, Andrés, David, Guille, Javi, Manu, Oscar and Rubén, who have shared with me many moments for more than 15 years. Thank you for your moral support, motivation and friendship, which made this journey more comfortable and drove me to give my best.

Finally, I would like to express my deepest gratitude to my family for their commitment, their unconditional love, their support and their infinite patience. Especially Jenifer, who closely shared with me joys and celebrations during this thesis, but also had to keep up with me during the endless periods of stress that precede academic deadlines. Thank you for your patience, comprehension, and support during these years of thesis.

[1]This work has been supported by the Spanish State Research Agency under grants TIN2013-44375-R, TIN2016-75344-R (AEI/FEDER, EU), and BES-2014-068225.

Contents

1 Introduction **25**

1.1 Current Trends . . . 25

1.1.1 Real Time Mobile Graphics Software . . . 26

1.1.2 Mobile Graphics Hardware . . . 28

1.2 Problem Statement . . . 30

1.2.1 Major Energy Consumers . . . 31

1.2.2 Major GPU Energy Consumers . . . 33

1.2.3 Occlusion Culling . . . 34

1.2.4 Collision Detection . . . 35

1.3 State of the art . . . 35

1.3.1 Reduction of Redundant Fragment Shading . . . 35

1.3.2 Collision Detection . . . 39

1.4 Thesis Overview and Contributions . . . 42

1.4.1 Visibility Rendering Order . . . 42

1.4.2 Render Based Collision Detection . . . 44

1.4.3 Other contributions . . . 46

2 Background **49**

2.1 Graphics Rendering Pipeline . . . 49

2.1.1 Application Stage . . . 49

2.1.2 Geometry Stage . . . 50

2.1.3 Rasterization . . . 55
2.2 GPU Microarchitecture: . . . 59
2.2.1 Immediate Mode Rendering . . . 59
2.2.2 Tile Based Rendering . . . 60

3 Methodology **63**
3.1 Simulators . . . 63
3.1.1 GPU Simulation . . . 63
3.1.2 Collision Detection CPU Simulation with Marss86 and Bullet . . . 67
3.2 Benchmarks . . . 68
3.2.1 Benchmarks Set . . . 69
3.2.2 Benchmarks Characterization . . . 72

4 Visibility Rendering Order: Improving Energy Efficiency on Mobile GPUs through Frame Coherence **77**
4.1 Visibility Determination and Overshading . . . 79
4.2 Visibility Rendering Order . . . 81
4.2.1 Overview . . . 81
4.2.2 Graph Generation . . . 82
4.2.3 Sort Algorithm . . . 83
4.2.4 Heuristics to Sort the Objects in a Scene . . . 85
4.2.5 Partial Order of Objects . . . 87
4.2.6 Visibility Rendering Order Adjustments . . . 88
4.3 Microarchitecture . . . 88
4.3.1 Deferred Rendering TBR GPU . . . 88
4.3.2 Visibility Rendering Order TBR GPU . . . 90
4.4 Experimental Framework . . . 96
4.4.1 GPU Simulation . . . 98
4.5 Experimental Results . . . 98

4.5.1 Effectiveness of VRO . . . 98
4.5.2 Overshading with Different Heuristics to Break Graph Cycles . . . 103
4.6 Conclusion . . . 104

5 Render-Based Collision Detection for CPU/GPU Systems 105
5.1 Collision Detection . . . 105
5.1.1 Image Based Collision Detection . . . 106
5.1.2 Enabling RBCD in the GPU . . . 108
5.2 Microarchitecture . . . 109
5.2.1 RBCD Overview . . . 109
5.2.2 Identification of Collisionable Objects . . . 109
5.2.3 Deferred Face Culling . . . 111
5.2.4 Insertion into the Z-depth Extended Buffer . . . 111
5.2.5 Z-Overlap Test . . . 112
5.2.6 Animation Loop . . . 113
5.2.7 Power model of the RBCD unit . . . 115
5.2.8 CPU Collision Detection Simulation . . . 115
5.3 Experimental Results . . . 115
5.3.1 Performance and Energy Consumption Benefits . . . 116
5.3.2 GPU Overheads . . . 118
5.3.3 Sensitivity to ZEB List Length . . . 122
5.4 Conclusions . . . 123

6 Conclusions 125
6.1 Conclusions . . . 125
6.2 Future Work . . . 128

Appendices 131

A Visibility Rendering Order on IMR GPUs **133**

A.1 Immediate Visibility Rendering Order . . . 133

A.1.1 Visibility Rendering Order Adjustments . . . 135

A.1.2 Visibility Rendering Order IMR GPU . . . 136

A.2 Experimental Framework . . . 137

A.3 IMR-VRO Results . . . 137

A.3.1 Software Z-Prepass . . . 142

A.4 Conclusions . . . 146

List of Tables

3.1 CPU Simulation Parameters. 68

3.2 Benchmarks Set. 73

3.3 Geometry Stage Stats. 73

3.4 Rasterization Stage Stats. 74

4.1 VRO alternatives. 87

4.2 GPU Simulation Parameters. 97

5.1 CPU/GPU Simulation Parameters. 116

5.2 Benchmarks. 117

5.3 Percentage of fragment overflow for a ZEB with 4, 8 or 16 entries (each entry holds data for one fragment). 122

A.1 Benchmarks. 137

A.2 GPU Simulation Parameters. 138

List of Figures

1.1 Millions of units of smartphones and PCs shipped. Source: Gartner [5, 7, 9, 14]. * Projected data. 26

1.2 Monthly OS Market Share from June 2012 to June 2017. Data provided by Statcounter Global Stats [19]. 27

1.3 Mobile Graphics Hardware Market Share. Data provided by Unity, March 2017 [11]. 29

1.4 Battery Capacity of Samsung Galaxy S smartphone series. 31

1.5 Total Power Consumption, GPU load and CPU load of a mobile device. Measurements made with Trepn Power Profiler [35, 33], with special features for Snapdragon SoCs. The phone employed in the two tests is a Samsung Galaxy J5, equipped with a 720x1280 (5") Super AMOLED display (294 ppi) and powered by a 28nm Qualcomm Snapdragon 410 MSM8916 SoC [34], which includes a 64 bit quad-core 1.2 GHz Cortex-A53 CPU and an Adreno 306 GPU. Both tests were done with screen brightness set to the minimum and WiFi disabled (cellular data enabled). 32

1.6 GPU energy breakdown and main memory BW breakdown of a mobile TBR GPU. Numbers obtained with the set of benchmarks introduced in Section 3.2. 34

1.7 Simplified version of the Graphics Pipeline. 42

2.1 Conceptual stages of the Graphics Rendering Pipeline. 50

2.2 Vertex-level transformations in the Graphics Rendering Pipeline. (1) The vertices of the 3d model are in Object Coordinates. (2) The vertices are scaled, rotated and translated to World Coordinates. (3) The vertices are positioned in the camera scope transforming them to Eye Coordinates (see detail of camera in orange outline). (4) The vertices transformed to Clip Coordinates by projecting them onto the near clip plane. (5) Perspective correction is applied to transform vertices to Normalized Device Coordinates. (6) Viewport transform is applied to translate vertices to Viewport Coordinates. Car 3D model courtesy of Alexander Bruckner [24]. 51

2.3 Perspective (left) and orthographic (right) viewing volumes. 52

2.4 OpenGL normalized viewing volume. 52

2.5 Examples of common topologies used both by OpenGL and DirectX [167]. 53

2.6 View-frustum including viewport. 54

2.7 Clipping cases for triangles (top), lines (middle) and points (bottom). 54

2.8 Clock wise and counter-clock wise winding triangles. 55

2.9 3D model of a sphere (top-left), detail showing the effect of face culling (top-right), wire-frame view of the sphere without face culling (bottom-left), wire-frame of the sphere with face culling enabled (bottom-right). 56

2.10 Sub-stages of the Rasterization stage of the Graphics Pipeline. In this example, a red triangle is rasterized over a blue background, and some fragments of the red triangle are occluded by a green triangle. 56

2.11 Edge functions (E01, E12 and E20) of a primitive defined by vertices v0, v1 and v2. 57

2.12 Detail of scene where three objects A, B and C are rendered in C, A, B order. The image depicts the fragments of every object that pass the Depth Test. 58

2.13 Detail of scene where two objects are blended. The image depicts how the fragments of a translucent object are blended with the colors already stored in the Color buffer providing a transparency effect. 59

2.14 Microarchitecture of an IMR GPU. 60

2.15 Microarchitecture of a TBR GPU. 61

2.16 Microarchitecture of a TBR GPU implementing Deferred Rendering. 62

3.1 Overview of Teapot simulation infrastructure. 64

3.2 NVIDIA Tegra like architecture (left), Mali 400 MP like architecture (right). Images from teapot paper [111]. 66

3.3 Overview of Marss components. Source: www.marss86.org. 68

3.3 This figure shows a screenshot for each of the Android games included in our set of benchmarks. 71

4.1 Simplified version of the Graphics Pipeline. 78

4.2 Shaded fragments per pixel in a GPU without Early-depth test, with Early-depth test and with perfect front-to-back rendering order at object granularity. 78

4.3 Graphics pipeline: (a) Sequential DR. (b) Parallel DR. 81

4.4 Graphics pipeline including VRO. 82

4.5 Visibility Graph generation for the given scene. 83

4.6 Sorting a Visibility Graph with the Kahn's algorithm. 83

4.7 Two example cases where object B sits in front of A and C. The shaded region highlights the overlap between A and C. 87

4.8 Raster Pipeline of a TBR GPU implementing Deferred Rendering. 89

4.9 Raster Pipeline of a TBR GPU implementing VRO. 90

4.10 Detail of Tile Engine structures involved in Geometry Fetching. 91

4.11 Detail of an entry of the Graph Buffer. Each entry is 512 bits (including 11 bits of padding). 92

4.12 Size of the Graph Buffer for different number of children nodes (W) and different number of maximum objects (from 8192 to 131072). 93

4.13 (Top) Nodes per frame, edges per frame and maximum number of nodes in an adjacency list. (Bottom) 75th, 85th and 95th percentiles of the size of the adjacency-lists of the scene graphs analyzed. 94

4.14 Edge Insertion Hardware. 95

4.15 Visibility Sort Hardware. (a) Initial search (b) Iterative procedure 95

4.16 Speed-up of DR and VRO normalized to the baseline TBR GPU. 99

4.17 Energy consumption of DR and VRO normalized to the baseline TBR GPU. 99

4.18 Overshading of DR and VRO normalized to the overshading of the baseline TBR GPU.100

4.19 Number of cycles to read a primitive with DR and VRO. 101

4.20 Normalized memory traffic of DR and VRO with respect to baseline GPU. 101

4.21 Increment of cycles reading geometry (first bar), increment of stall cycles caused by the Fragment Processing stage (second bar), and increment of cycles of execution of the Raster Pipeline (third bar) all using DR with respect to VRO. 102

4.22 Energy breakdown for the system Main-Memory/GPU with DR (left) and VRO (right) both normalized to the baseline GPU. 103

4.23 Normalized Overshading with different Heuristics to break Graph Cycles. 104

5.1 Discretized representation of the entry and the exit points of the surfaces in a 3D scene for pixels P1, P2 and P3. The Y-axis is a one-dimensional representation of the projection plane and the Z-axis represents depth. 107

5.2 (a) Front view of a 3D scene (b) AABBs as collisionable shapes (c) Convex hull for GJK algorithm (d) RBCD. 108

5.3 GPU microarchitecture including an RBCD unit. 110

5.4 Sorted insertion hardware. 112

5.5 Interference cases between front-faces ([) and back-faces (]) of two objects, A and B. 113

5.6 Z-overlap Test hardware. 114

5.7 Example of game loop execution in the CPU/GPU system, (a)without RBCD, (b)with RBCD. CR and GCI stand for Collision Response and GPU Command Issue respectively.114

5.8 (a) RBCD speedup regarding Broad-CD, (b) Normalized energy consumption of RBCD regarding broad-CD, (c) RBCD speedup regarding GJK-CD, (d) Normalized energy consumption of RBCD regarding GJK-CD 117

5.9 (a) Normalized rendering time of the GPU with RBCD w.r.t. the baseline GPU. (b) Normalized energy of the GPU and main memory with RBCD w.r.t. the baseline GPU.119

5.10 (a) Normalized energy consumption of the GPU with RBCD w.r.t the baseline GPU. (b) Normalized main memory energy of RBCD w.r.t. the main memory energy of the baseline GPU. 120

5.11 Energy GPU/Main memory breakdown. 120

5.12 GPU time breakdown including time of Geometry and Raster pipelines. 120

5.13 Tile Cache loads, primitives, fragments, and Raster Cycles with the GPU including RBCD normalized to the GPU baseline. 121

6.1 Simplified version of the Graphics Pipeline. 125

A.1 Memory bandwidth usage on a mobile GPU for a set of commercial Android games. On average 98.5% of the bandwidth to main memory is caused by operations performed after rasterization. 134

A.2 Graphics pipeline including VRO. 134

A.3 Baseline IMR GPU architecture (left) and IMR GPU architecture including VRO (right). 136

A.4 Speed-up of IMR-VRO normalized to the baseline IMR GPU. 139

A.5 Energy consumption of IMR-VRO normalized to the baseline IMR GPU. 139

A.6 Energy breakdown of IMR-VRO normalized to the baseline IMR GPU. 140

A.7 Overshading and Instructions Executed in the Fragment Processors with IMR-VRO normalized to those of the baseline IMR GPU. 140

A.8 Ratio between the time savings and the time overhead of VRO (higher is better). . . 141

A.9 Energy Delay Product of IMR-VRO normalized to the baseline IMR GPU (lower than 1 is better). 141

A.10 Memory bandwidth usage of IMR-VRO normalized to baseline IMR GPU. 141

A.11 Activity factors with IMR VRO normalized with those of the baseline IMR GPU for Color cache (top left), Depth cache (top right), Texture Caches (bottom left) and L2 cache (bottom right). 142

A.12 Simplified version of the Graphics Pipeline executing Z-Prepass. 143

A.13 Speed-up of Z pre-pass normalized to the baseline IMR GPU. 143

A.14 Energy consumption of Z pre-pass normalized to the baseline IMR GPU. 144

A.15 Energy Delay Product of Z pre-pass normalized to the baseline IMR GPU (lower than 1 is better). 144

A.16 Memory bandwidth usage of Z pre-pass normalized to baseline IMR GPU. 144

A.17 Speed-up of VRO and Z pre-pass normalized to the baseline IMR GPU (one FP in all cases). 145

A.18 Energy consumption of VRO and Z pre-pass normalized to the baseline IMR GPU (one FP in all cases). 146

A.19 Energy Delay Product of VRO and Z pre-pass normalized to the baseline IMR GPU (lower than 1 is better, one FP in all cases). 146

1

Introduction

This chapter introduces the main issues in the architectural design of mobile GPUs and how they have evolved over the years. Then, we present the specific problems we approach in this thesis, how these problems have been addressed by other authors and, finally, our proposals to solve them.

1.1 Current Trends

In recent years, mobile devices have become powerful and ubiquitous computational engines. They have quickly incorporated graphics and animation capabilities that in the 1980s were only seen in industrial flight simulators and a decade later became popular in desktop computers and game consoles. At each new generation, the growing computing power of mobile devices has promoted the adoption of more powerful support for graphics and real-time physics simulations in all mobile devices, with increasing precision and realism, which does not seem to slow down in the near future. Furthermore, with the rise of Mobile Augmented Reality and Mobile Virtual Reality applications it is expected that the consumers will keep demanding an increasing degree of realism in interactive rendering of images and at higher rendering resolution.

Nowadays, among other uses, we can surf the Internet at high speed, play 3D games, reproduce and record HD video and take high resolution photographs. The evolution of the mobile device from being just a device to make and receive voice calls to being a multimedia and multitasking device, has led the phone to surpass computers for the leisure time we expend in multimedia. As Figure 1.1 shows, sales of mobile devices have dramatically increased from being around 0.97 billion units in 2013, to pass 1.2 billion units in 2014 and reach more than 1.9 billion units in 2015. In 2016 the number of unit shipped decreased, but it is expected to grow in following years. At the same time, the number of downloaded applications was estimated to be above 100 billion in 2013 and it

is expected to reach around 268 billion in 2017 [3]. Furthermore, the rise of Internet-of-Things and the introduction of Virtual and Augmented Reality may further increase this tremendous growth rate in the near future.

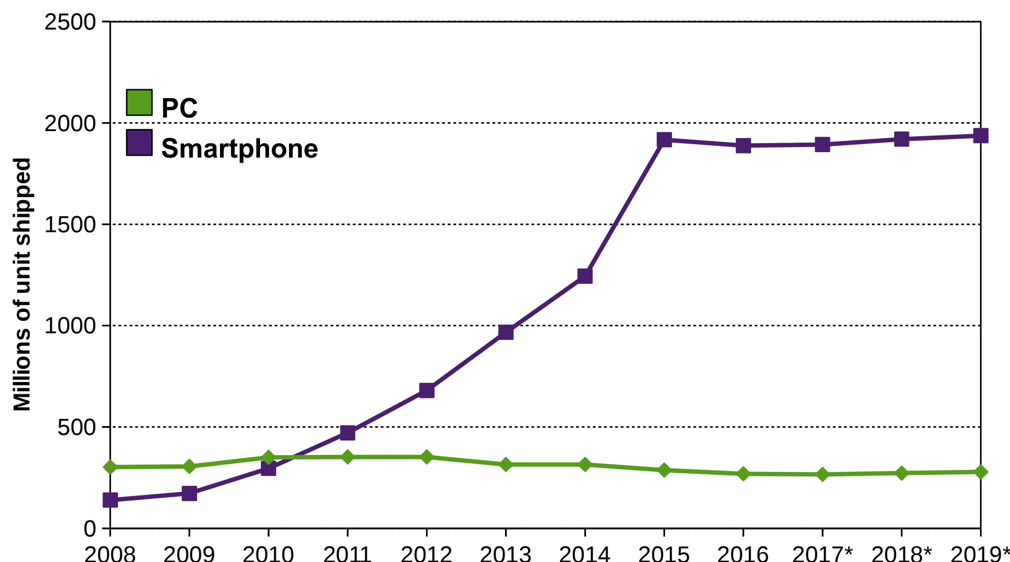


Figure 1.1: Millions of units of smartphones and PCs shipped. Source: Gartner [5, 7, 9, 14]. * Projected data.

1.1.1 Real Time Mobile Graphics Software

In this section we briefly review the software stack of mobile devices including Operating Systems, applications, Game Engines, Physics Simulation and graphics APIs. The Operating System (OS) is the software that provides services to the applications in current mobile devices. The OS allows to access the rich variety of hardware available in the system, like the screen, the microphone, the speakers, the accelerometer, the GPS, the camera, or the GPU. Note that in current OSs, even the user interface uses GPU acceleration. Android [10] and iOS [12] are the Mobile OSs that have become prevalent with around 72% and 20% of the Mobile OS market share. Moreover, as Figure 1.2 shows, since April 2017 the OS market share of Android (39%) is even bigger than the market share of Windows (37%).

Android is an open source software stack based on Linux, which is meant to work on a great variety of devices (tablets, smart watches, televisions, even phones) from multiple vendors (Samsung, Google, LG, Huaweii, Sony, BQ, etc). Android provides numerous layouts that allow the programmer to add different pre-compiled objects in order to create an interface that provides a rich user experience. However, the developer is responsible for adequately employing Android's resources to deliver applications that provide a flexible user interface to be compatible with different devices.

Regarding applications, there is a plethora of mobile applications available on on-line stores like Google Play Store [44] and Apple App Store [51], being 3D and 2D graphics animation applications the most popular ones [15, 13]. To develop such applications it is common to employ a game engine,

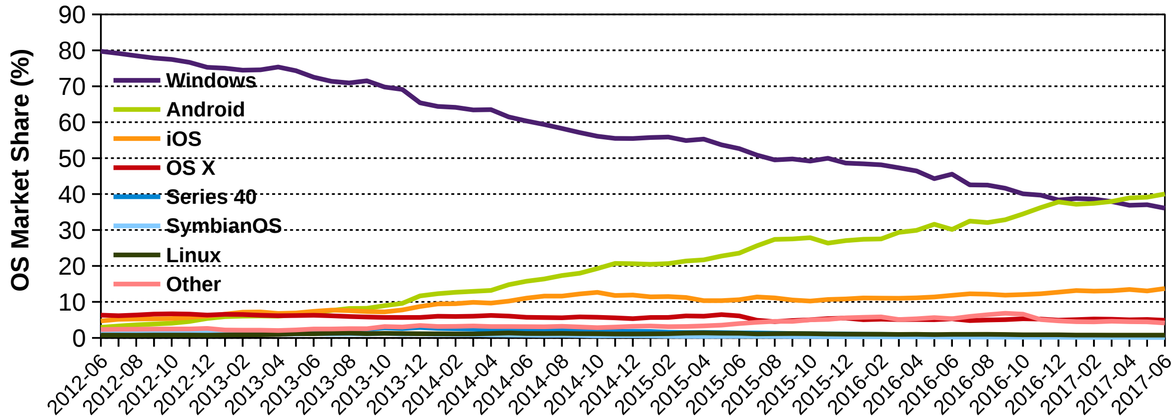


Figure 1.2: Monthly OS Market Share from June 2012 to June 2017. Data provided by Statcounter Global Stats [19].

a SDK that usually includes functionality for graphics rendering, physics simulation, animation, artificial intelligence and sound, among others. Some of the most popular game engines are Unity [21] and Unreal Engine [22], and both of them offer an optimized solution for mobile devices that allows to create 2D and 3D mobile games for iOS and Android.

Regarding physics simulation, the physics engine is the software responsible for providing realistic physical behavior to the objects in a scene, whose velocity and direction must be affected by collisions with other objects and by gravity and other forces. Bullet [129] stands out as an open source framework which simulates collision detection and collision response between objects. Bullet can be used in the Unreal Engine, even though it includes its own physics simulation solution (Physics Assets Tool). Likewise, Unity supplies its own 2D and 3D physics engine as well as a plugin to use Bullet.

Regarding graphics rendering, the game engine offers a rendering method that handles the rendering of a given scene, being an intermediary between the application and the graphics API. The graphics API provides a low-level yet abstract way to access the specific graphics hardware accelerator included in the device. On desktop, the most common real time 3D graphics APIs are OpenGL and DirectX from the Khronos Group [52] and Microsoft respectively [42]. On the mobile side, both iOS and Android include native support for embedded graphics acceleration through OpenGL ES [57], which is a cross-language and multi-platform API that has gone through several versions since its first specification was released in July 2003. Although OpenGL ES 1.0 (based on OpenGL 1.3) is mostly meant to enable software renderers, it also enables basic hardware acceleration. Following versions enable advanced hardware acceleration in order to increase performance and reduce memory bandwidth to save energy. OpenGL ES 1.1 [16] (based on OpenGL 1.5), is meant for fixed-function hardware renderers (configurable but not programmable), and versions 2.X and 3.X enable full programmable 3D graphics by replacing most fixed-function rendering pipeline stages by programmable versions [58, 17].

1.1.2 Mobile Graphics Hardware

Current mobile devices employ SoCs, which are heterogeneous computing systems where the key to enable a low-power yet rich experience is to utilize the right accelerator for every task. The leading industry vendors of SoCs for mobile devices have already integrated new custom designed accelerators such as: Digital Signal Processors; Display, Connectivity and Security controllers; Image Signal Processors (camera support); Video Encoder/Decoders; and, of course, the GPU. In order to further improve energy efficiency, some vendors have already included special cores even into the GPU, like the 2D graphics composition cores [68, 23]. These special hardware accelerators either replace a software implementation that was previously executed in the general purpose hardware CPUs, or expand the capabilities of the SoC. In this section we briefly review the evolution of the main components of the graphics system of a mobile device: the resolution of the display technology and the graphics acceleration hardware.

Regarding the display technology, we have seen an impressive increment in the display resolution as well as the color depth [8]. Back in 2004 the first licensed OpenGL ES 1.0 device, the Imageon 2300 of ATI Technologies, offered 320 x 240 display resolution and 16 bpp of color depth. Today, resolutions of 1920 x 1080 (Full HD) and 32 bpp are quite common and available in devices like iPhone 7 Plus and Samsung Galaxy S5. Other devices like Samsung Galaxy S7 increase the display resolution to 2560 x 1440 (QHD) in 5.1 inches, while Samsung Galaxy S8 goes up to 2960 x 1440 (QHD+) native resolution in 5.8 inches. Regarding mid-end devices, the most common resolution is 1280 x 720 (HD Ready).

At the same time that the screen resolution has increased, the mobile graphics rendering has significantly evolved from being responsible for creating simple text user interfaces mostly implemented in software and executed in the CPU, to creating complex 2D/3D graphics animations that employ a graphics hardware accelerator, the Graphics Processing Unit (GPU). In fact, in many cases the early mobile devices had only hardware support for integer arithmetic [59]. However, with the gradual adoption of higher screen resolutions and the necessity to render high-quality images the graphics system was required to provide higher fill-rates, and subsequently GPUs became an essential requirement.

The Khronos Group consortium maintains a list of products conformant with the different OpenGL ES specifications [18]. According to this list, the first mobile GPUs to support OpenGL ES 1.1 were the Mali-110/Mali-55 [1] in 2005. They were fixed-function hardware pipelines (configurable but non-programmable) for rasterization and Fragment Processing, offering fill rates in the order of 100 Mpix/sec [89]. Although the Geometry Processing might still be performed by software running on the CPU, an optional MaliGP (Geometry Processor) processor could also be added to offload the CPU of the Geometry Processing and enable 3D applications to run faster [190].

The first mobile GPUs to support OpenGL ES 2.0 arrived in 2008 and were PowerVR SGX530 and Mali200, offering fill rates in the order of 200 Mpix/sec [53, 60]. OpenGL ES 2.0 replaced some fixed-function hardware stages of the pipeline by programmable stages with the ability to execute vertex and fragment shader programs written in OpenGL ES Shading Language, to improve energy efficiency. The first mobile GPU to officially support OpenGL ES 3.0 was the Adreno 320 [37] presented in 2013 in the Snapdragon 600 SoC of Qualcomm [67], with a reported fill rate of 1.6 Gpix/sec and able to process 225 Mtri/sec. Following specifications of OpenGL made a big step

towards programmability. Compute Shaders were added in OpenGL ES 3.1, while Geometry and Tessellation Shaders were presented in OpenGL ES 3.2. The former allows to perform computation not directly related to drawing triangles and fragments, while the latter allows to efficiently process complex scenes on the GPU by means of giving the programmer the opportunity to create new geometry not presented in the input stream of vertices. The first GPU to support OpenGL ES 3.1 is the Qualcomm Adreno 510 in 2016. The first GPU to comply with OpenGL ES 3.2 is the ARM Mali-G71 also in 2016 with a fill rate around 1850 Mtri/sec and 27.2 Gpix/sec.

Regarding the current state of the market of Mobile Graphics Hardware (see Figure 1.3), it is worth mentioning that it is not dominated by a single vendor. However, ARM and Qualcomm are clearly ahead of the other ones hoarding 44% and 35% of the market share respectively, followed by the duo formed by Apple and Imagination Technologies with more than 16% of the market (8.5% and 7.8% respectively). The list is closed by Vivante (1.8%), Broadcomm (0.9%), NVIDIA(0.6%) and Intel(0.5%).

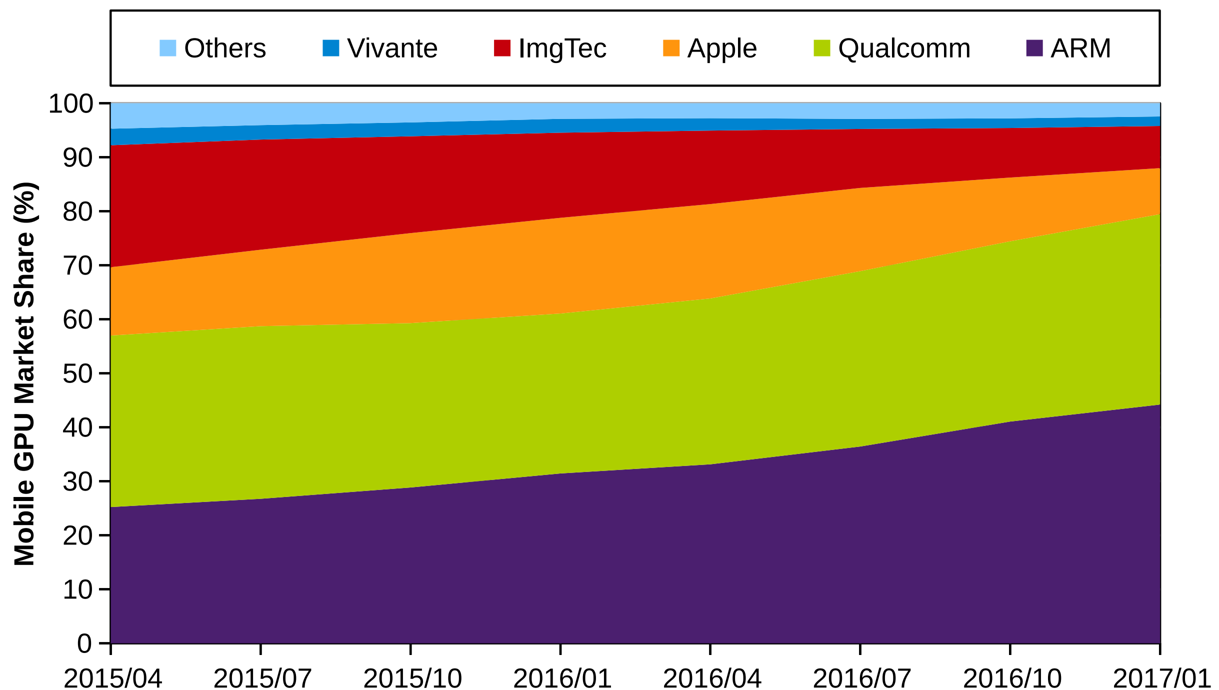


Figure 1.3: Mobile Graphics Hardware Market Share. Data provided by Unity, March 2017 [11].

ARM licenses the Mali family of GPUs, with three main lines of GPUs: Ultra-low power (Mali 400 series), high area efficiency (Mali 720) and high performance (Mali G71, T860/880) [55]. The Mali 400 MP [54] is the most deployed Mali GPU with around 19% of the market. It includes OpenGL ES 1.1/2.0 support and is able to process 55 Mtri/sec and 2.0 Gpix/sec. The Mali400 MP is a multi-core Tile Based Renderer, i.e., the rendering view-port (screen) is divided into tiles (bins), which are independently rendered tile by tile using on-chip buffers that minimize main memory bandwidth usage. This architecture is explained in more detail in chapter 2. A Mali G71 GPU can be found in popular smartphones like Samsung Galaxy S8 and S8+, powered by the Samsung

Exynos SoC [65, 63].

Adreno is a series of mobile GPUs produced by Qualcomm included in the Snapdragon SoC [38, 69]. Rather than implementing a Tile Based or an Immediate renderer, Adreno GPUs implement both and are able to switch between the two modes at run-time using the FlexRender technology [43]. Adreno GPUs can be found on commercial devices like Samgung Galaxy S5 [64] or HTC One M9 [47].

PowerVR [61] is the solution provided by Imagination Technologies for embedded devices. PowerVR implements a Tile Based Deferred architecture, which implements Hidden Surface Removal at pixel granularity before computing the pixel colors, i.e., it defers the computation of the pixel colors until the visibility for a pixel is determined. This approach decreases the Fragment Processor computations and further reduces main memory bandwidth. PowerVR GPUs can be found in several Mediatek SoCs, like the Helio x10/x30 [45, 46]. Furthermore, PowerVR designs are also included in Apple SoCs like the Apple A8, Apple A9 and Apple A10 [40, 41, 39], which are employed in popular mobile devices like iPhone 6/6 Plus, iPad Pro and iPhone 7/7 Plus [49, 48, 50]. The Playstation Vita of Sony includes a PowerVR SGX543MP4+ GPU [62].

NVIDIA delivers the Tegra SoC [70], which includes an immediate-mode rendering GPU architecture specially designed and tuned for high-performance and power efficiency. Tegra X1 implements a Maxwell GPU architecture with 256 cores, being the most advanced model that NVIDIA provides for mobile devices [71]. It can be found on devices like the Nintendo Switch game console [56] and the NVIDIA SHIELD TV streaming player [66].

Finally, it is worth to mention a vendor whose GPUs are not so widespread but, like the previous companies, it offers high-quality products that achieve similar performance and power consumption. Vivante develops the Vega GPU architecture included in the GC family of processors [23], which counts with more than hundred successful mass market SoC designs. Samsung Galaxy Ace Plus, Huaweii Ascend P6 and Samsung Galaxy Tab 4 are examples of mobile devices powered by Vega [72].

1.2 Problem Statement

In this section we first illustrate the importance of energy-efficient designs to improve battery life per charge and performance in mobile devices. Then, in subsections 1.2.1 and 1.2.2 we point out the main sources of energy drain in mobile SoCs for graphics workloads. Finally, in subsections 1.2.3 and 1.2.4 we introduce the two main aspects focused in this thesis.

Software and hardware improvements like the ones described in the previous section are crucial to deliver a rich user experience that satisfies the user demands on the functionality of mobile devices. For example, in graphics animation applications the trend is towards supporting more realistic and complex 3D rendering. The direct consequence of supporting these higher computing demands comes with a significant increase in energy consumption. However, the battery capacity does not grow at the same pace as the computing demands, which produces an energy gap that is incremented on each generation [168] and puts pressure on the battery life of mobile devices. It is important to note that for the users of mobile devices the battery life per charge is the third most

valued satisfaction factor [32], being critical in the overall satisfaction with the device [31]. Figure 1.4 shows the evolution of the battery capacity of the Samsung Galaxy S family of smartphones. As can be seen, the battery capacity has only been doubled from 2010 to 2017 with an average increment of 11.7% per year. In particular, since 2014 -three generations ago-, the battery capacity of the flagship device of Samsung has only grown slightly above 7%, remaining flat in the last two generations.

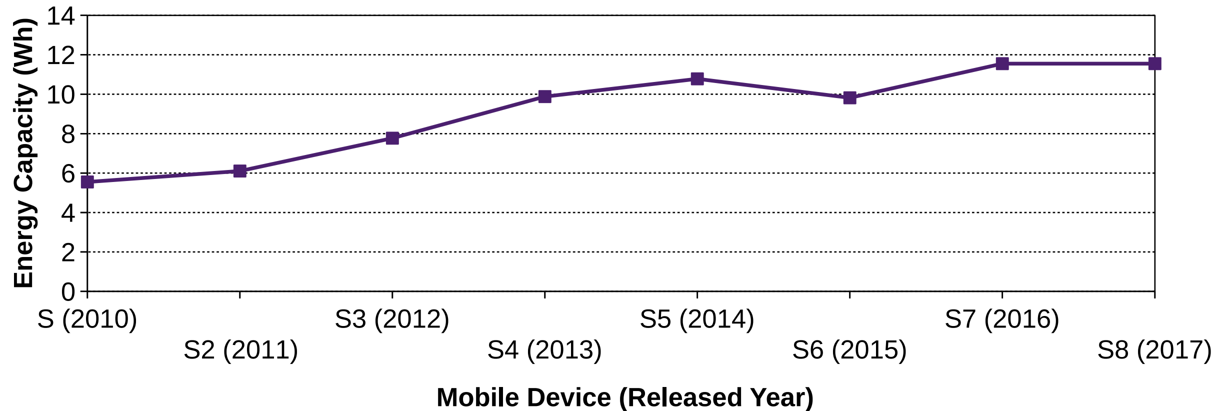


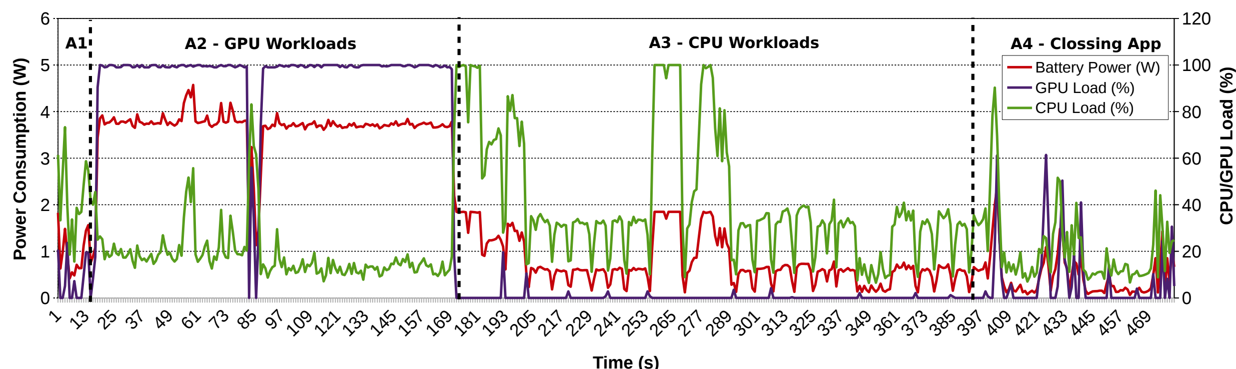
Figure 1.4: Battery Capacity of Samsung Galaxy S smartphone series.

Besides battery life per charge, there is another factor that limits energy consumption on current mobile devices. Current mobile devices like smartphones and tablets are extremely thin in order to reduce their weight, as well as make them more attractive and easier to slip into pockets of all sizes. As a direct consequence, these devices cannot be equipped with active cooling mechanisms like fans or conventional and big heat sinks to dissipate heat in order to keep the temperature below the maximum temperature limit of internal components. Moreover, given the handheld nature of these devices, it is critical to keep a comfortable surface touch temperature, otherwise the device may become too hot to handle [195, 102]. In order to avoid overheating above thermal limits, mobile SoCs tend to be throttled when the load is high for a large period of time [178, 85].

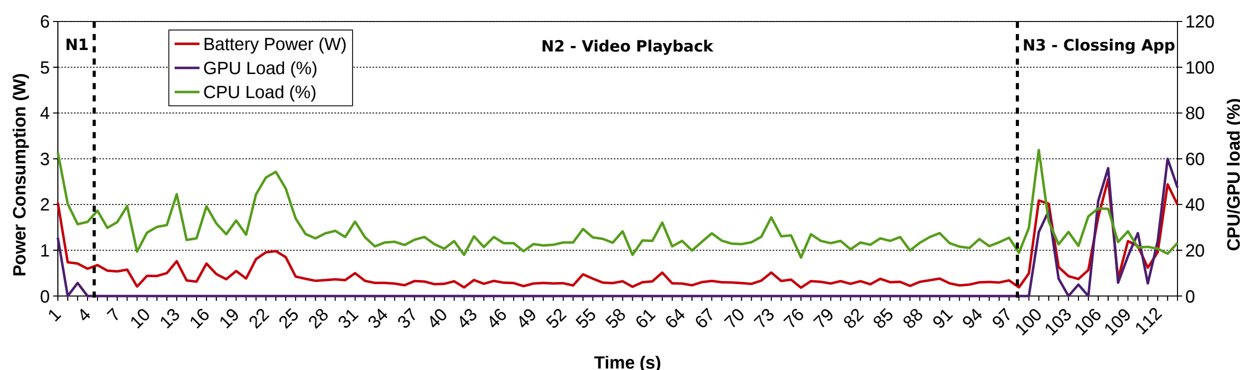
1.2.1 Major Energy Consumers

Among all the components of the mobile device the GPU and the CPU have been identified by previous works to be the principal energy consumers for common applications [122]. In particular, for graphics applications the GPU has been identified as the principal energy consumer [174]. Further experimental data with the same SoC shows a peak consumption of the GPU 50% higher than the peak consumption of the CPU [30]. We conduct a brief experiment in order to further motivate our work.

Figure 1.5 shows the total power consumption of a smartphone (left axis) as well as the GPU and the CPU load (right axis) for two different applications: AnTuTu [36]; and Netflix [81] for Android. AnTuTu, with more than 100 M users, is the most popular benchmarking application for mobile devices powered by Android. Netflix is a popular video streaming platform. Let us analyze in detail the results for AnTuTu benchmark. Figure 1.5a shows the results for AnTuTu benchmark.



(a) Full run of AnTuTu [36] benchmark V6.2.7.



(b) Netflix HD video playback using Cellular Data.

Figure 1.5: Total Power Consumption, GPU load and CPU load of a mobile device. Measurements made with Trepn Power Profiler [35, 33], with special features for Snapdragon SoCs. The phone employed in the two tests is a Samsung Galaxy J5, equipped with a 720x1280 (5") Super AMOLED display (294 ppi) and powered by a 28nm Qualcomm Snapdragon 410 MSM8916 SoC [34], which includes a 64 bit quad-core 1.2 GHz Cortex-A53 CPU and an Adreno 306 GPU. Both tests were done with screen brightness set to the minimum and WiFi disabled (cellular data enabled).

The test has been divided into four phases: A1: application startup; A2: GPU workloads; A3: CPU workloads; A4: application end. As can be seen, the peak power consumption (around 4.57 W) is reached when running the graphics workloads (A2 phase). Furthermore, the average power consumption on this phase is around 3.7 W. Note that the power consumption correlates well with the GPU load, which is around 96% on this phase. On GPU workloads phase the CPU load is around 18.9%. Regarding the CPU workloads (A3 phase), the average power consumption is around 0.78 W with an average CPU load of 40%. The average GPU load is smaller than 1% in this phase. Note that even when the CPU load is almost 100% the total power consumption is still below 2 W. These results suggest that the GPU is the principal energy consumer in this mobile device. However, the reader may argue that the figure not only shows the power of the GPU and the CPU, but also the power of all the components of the smartphone such as the screen and the modem, and therefore they could be the principal power consumers in the device. In order to

elucidate the power consumption of the other parts of the device, like the modem and the screen, we have run an additional test playing a streaming video with Netflix (see Figure 1.5b). The span of this figure is split in three different phases: N1: application startup); N2: video playback; N3: application end. When playing the video, the peak energy consumption is right below 1 W, while the average is around 0.36 W. The GPU load is 0%, while the CPU load is around 26% on average, with a maximum of 56%. Note that, like in the previous test, when the GPU is idle (N2) the power consumption is much lower than when the GPU is busy (N3). The GPU load peaks that appear both in A4 and N4 are caused by the activity of the OS, which employs GPU acceleration for desktop transitions and animations. It is clear that the energy consumption of graphics workloads is much greater than the energy consumption of other workloads.

Let us analyze the battery life of a device when running GPU and CPU phases shown above. The Samsung J5 is equipped with a 2600 mAh battery at 3.8 V of nominal voltage, thus its battery capacity is: $\frac{2600mAh*3.8V}{1000} = 9.88Wh$. The amount of energy available is: $9.88Wh*3600s/h = 35568J$. As the average power drain is 3.7 W, 0.78 W and 0.36 W for A2 (GPU workloads), A3 (CPU workloads) and N2 (video playback) phases, the corresponding battery life is around 2 hours and 40 minutes, 12 hours and 40 minutes, and 27 hours and 26 minutes respectively. The battery life of the device when running the graphics workloads is heavily reduced being less than three hours, while for the non graphics workloads the battery life is more than ten hours.

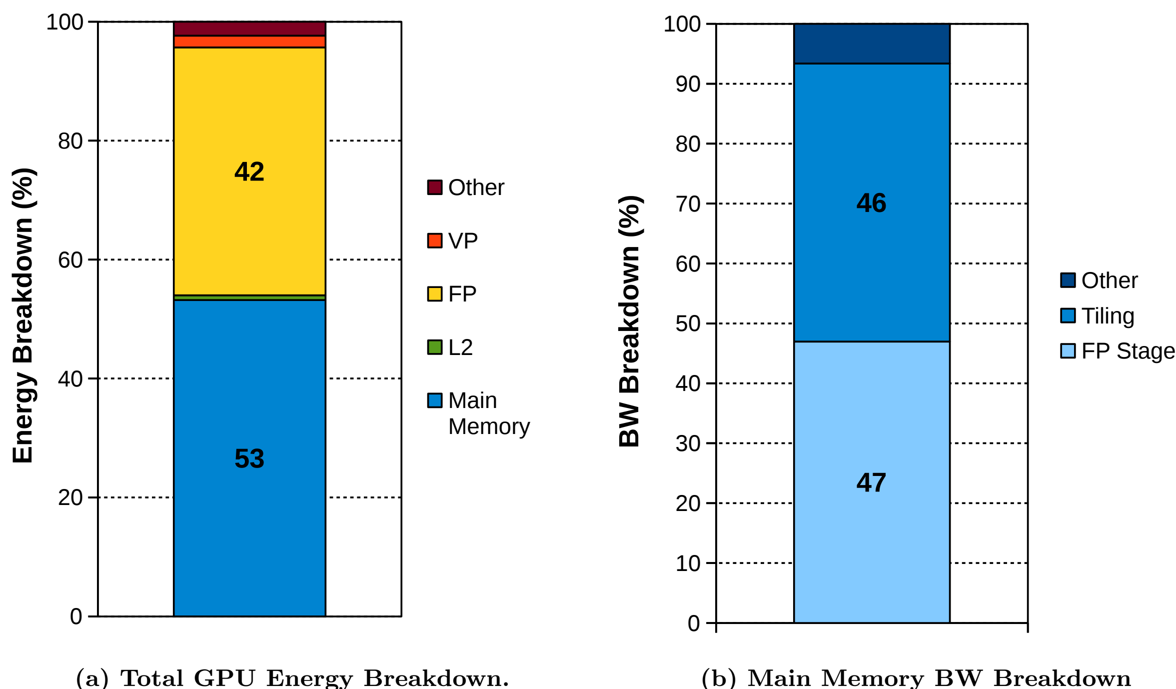
Given the huge impact of the graphics workloads and the GPU in the power consumption of mobile devices, the design of energy-efficient techniques for graphics workloads is crucial. Increasing the energy efficiency of the mobile GPU is not only meant to reduce energy consumption but also is an effective way to increase its performance, which makes mobile GPU designers focus on energy-efficient solutions to improve performance per watt rather than raw performance, which is key in making possible the increased computing demands of the users.

1.2.2 Major GPU Energy Consumers

Previous studies pointed out that the principal energy consumers in the graphics subsystem are the off-chip main memory accesses and the processors inside the GPU, in particular the ones devoted to Fragment Processing [102, 105, 165, 177]. The results obtained with our simulation infrastructure are in line with the results of the previous studies. Figure 1.6a shows the power of different structures of the GPU. The principal consumers are the accesses to main memory and the activity of the Fragment Processors (FP) with 53% and 42% of the total energy consumption respectively. The FPs are the main energy consumers mainly due to the fact that in common graphics workloads the number of fragments is usually orders of magnitude higher than the number of vertices. For example, for a rendering resolution of 1280 x 768 the number of instructions executed in the FPs is on average around 94% of the total instructions (numbers obtained with the simulation infrastructure and workloads described in the following chapter). Figure 1.6b shows the main memory BW breakdown originated at different stages of the graphics pipeline. As can be seen, around 47% of the BW with main memory is performed by the FP stage, so a large percentage of the energy consumption of the main memory corresponds to the FP stage.

Given that in our experiments around 53% of the energy breakdown corresponds to main

memory, and due to the fact that around 47% of the traffic is caused by the FP stage, the FP stage consumes around $53\% * 47\% = 25\%$ of the total energy performing accesses to the main memory. This 25% along with the 42% of the energy devoted to the FPs (see Figure 1.6a), means that the FP stage is responsible of around 67% of the total energy consumption of the main-memory/GPU system.



(a) Total GPU Energy Breakdown. **(b) Main Memory BW Breakdown**

Figure 1.6: GPU energy breakdown and main memory BW breakdown of a mobile TBR GPU. Numbers obtained with the set of benchmarks introduced in Section 3.2.

1.2.3 Occlusion Culling

During real-time graphics rendering, objects are processed by the GPU in the order they are submitted by the CPU, and occluded surfaces are often processed even though they will end up not being part of the final image. Occlusion culling (aka visibility determination) is an essential process of the graphics rendering pipeline that allows to decide which fragments of a scene are visible and which are not. The most widespread method to resolve visibility at pixel granularity is the Depth Test, which is typically placed at the end of the pipeline. The Depth test compares each fragment's depth against that already stored in the Depth Buffer to determine if the fragment is in front of all previous fragments at the same pixel position. If not, the fragment is discarded. Otherwise, the Depth Buffer is conveniently updated with the new depth, and the color of the fragment is sent to the blending stage, which will accordingly update the Color Buffer (the buffer where the image is stored).

The main advantage of the Depth test is that it ensures correct scene rendering regardless of the order the opaque geometry is submitted by the CPU. The main drawback is that the color of a

given pixel may be written more times than necessary (a problem known as overdraw), which wastes a considerable amount of main memory bandwidth and energy [172]. Moreover, when the GPU realizes that an object or part of it is not going to be visible, all activity required to compute its color has already been performed, with the consequent waste of time and energy. Hence, reducing the overshading produced by non-visible fragments can significantly increase the energy-efficiency of the GPU. As we will show, with some adaptations occlusion culling can be employed to reduce redundant fragment shading and therefore increase the energy-efficiency of the GPU.

1.2.4 Collision Detection

Collision Detection (CD) is one of the most important algorithms that manage the dynamics of the animations. It identifies the contact points between the objects of a scene and determines when they collide. During the past decade, mobile devices have quickly incorporated graphics and animation capabilities and the trend going forward is towards more powerful support for real-time physics simulations with increasing precision and realism. However, despite high-accurate physics algorithms increase the quality and the realism of the animations, their implementations may be expensive for mobile devices. Furthermore, highly-accurate Collision Detection schemes are control-intensive algorithms [2], known to cause branch divergence in GPGPUs [139] which produces low utilization of the functional units and reduces the performance [147] of the GPU.

1.3 State of the art

In this section we introduce techniques that other authors proposed to deal with occlusion culling and collision detection, the principal problems addressed in this thesis.

1.3.1 Reduction of Redundant Fragment Shading

Olson [172] studies the effect of overshading in mobile platforms for a set of commercial mobile applications and identifies it as a significant source of wasted energy. Given that the FP stage is the most energy-consuming stage of the graphics rendering pipeline, multiple works aim at reducing the number of instructions executed in the Fragment Processors by means of reducing the number of fragments whose color must be computed to render a frame. Some works make use of visibility determination to achieve some sort of reordering either at geometry or at fragment level granularity. There are other efforts that introduce changes in the GPU microarchitecture in order to reduce the number of executions of the Fragment Processors. On the other hand, rather than reducing the number of fragments executed in the processors, other studies have focused on improving the energy-efficiency of the GPU in different aspects, like hiding the memory latency or reducing the main memory BW of the GPU [109, 110].

Let us remark some concepts which will help us to better understand this section. When we refer to geometry of a scene we mean vertices and primitives (points, lines, and triangles) formed by those vertices. Such primitives are later discretized in fragments that correspond to pixel positions

and whose color is computed in the processors of the GPU. The fragments are usually grouped in 2x2 pixels regions called quad fragments, that represent the unit of execution of the Fragment Processors. All the quad fragments of an object (draw command, drawcall) execute the same program/shader.

Geometry Level Sorting

Multiple works have analyzed how overshading can be reduced by culling the geometry at primitive level granularity through the use of occlusion queries [118, 187]. When using occlusion queries, the application usually sends a query with a Bounding Volume of the object to the GPU where it is rasterized and depth tested for visibility. Eventually, the result of the query is sent to the driver and if the Bounding Volume was occluded, the application will not send the object to the GPU. Unfortunately, using any kind of feedback from the GPU limits the achievable frame rate unless the scene complexity is above a large threshold [117], which is often not the case on mobile workloads. Moreover, as mobile devices evolve to higher resolutions, testing for occlusion objects that are not fill-rate bound (i.e. with simple fragment programs and textures) may require many more pixels to fill and the GPU will likely spend more time rendering the object's bounding volume than the object itself. Given that at some point the application must know the result of the queries, they may introduce CPU stalls and produce GPU starvation. Furthermore, GPU drivers let the GPU render several frames behind the CPU by actually queuing the rendering commands [130, 92, 90], which exacerbates these problems. One important limitation when using occlusion queries is that it is required to sort the queries of the objects in a front-to-back manner to perform well. Many software approaches require building costly spatial hierarchical data structures to render the scene from any single viewpoint. They are quite effective on walkthrough applications where the entire scene is static and only the viewer moves through it, because the overheads can be amortized along a large number of frames [125].

Govindaraju et al. [141] sort the primitives of every object of a scene in a front-to-back order from a given viewpoint. However, they assume that the objects do not overlap, so the scheme only avoids overshading produced by geometry of the same object (intra-object overshading), and not between different objects (inter-object overshading). Unfortunately, this scheme is not able to handle cycles in the sorting process, which are highly common on dynamic 3D scenes.

Nehab et al. [171] and Sander et al. [185] also propose techniques to reduce intra-object overshading. They propose a pre-processing scheme that sorts the geometry of every object in a scene in a view-independent order that minimizes overshading. They focus on static meshes and produce a single order per mesh. On the other hand, Han et al. [146] propose a technique that is also focused on reducing intra-object overshading but they target animations. They propose a pre-processing scheme that produces a number of view-dependent orders per every object of a scene from different points of view, all of them minimizing the overshading. Then, the application selects at runtime a view-dependent order that minimizes the overshading depending on the orientation of the objects with respect to the camera.

Fragment Level Sorting

To reduce overshading, most graphics rendering pipelines perform an "early" visibility determination (Early-depth test) that tests the visibility of fragments before they are sent to the processors. Although the Early-depth can only cull fragments which are hidden by those already tested, it may reduce fragment shading and bring important performance and power benefits.

With Z-prepass, Haines et al. [145] address overshading by performing two separate rendering passes with the GPU. First Z-prepass renders the geometry without computing the color of the fragments, just using a null fragment shader in the processors, to determine the visibility of the scene. On a second pass with the real shaders the Early-depth test will perform optimal culling, so overshading will be minimum (just one opaque fragment per pixel will be shaded and written into the Color buffer, the buffer where the rendered image is stored). Unfortunately, this approach doubles several stages of the graphics pipeline like vertex processing, rasterization and visibility determination, which may offset the benefits of the technique. It is only effective for workloads with enough complexity where the overhead of the first rendering pass is compensated by large fragment computation savings, which is not usually the case on mobile applications.

Like Z-prepass, Deferred Rendering (DR) is a hardware technique that avoids overshading through computing the Depth Buffer before starting fragment shading. Imagination Technologies implements a Deferred Rendering approach in its PowerVR [91] family of GPUs. PowerVR implements a Tile Based Deferred Rendering (TBDR) architecture, a form of Tile Based Rendering (TBR) architecture. TBR pipelines divide the screen space into tiles of pixels. In a first phase before rasterization they perform all the geometry-related computations of the scene and sort the resulting geometry into screen tiles of fixed size (16x16, 32x32, 64x64 are common tile sizes). When all the geometry has been sorted and stored in the Parameter Buffer a second phase reads the geometry from it and renders the scene tile by tile in an independent manner. This allows the GPU to use small on-chip memories to contain the Depth and the Color buffers for the entire tile, which dramatically reduces the accesses to main memory [106]. Note that the overshading still occurs, but the accesses to Depth and Color buffers are performed over local and small on-chip memories, instead to main memory. Additionally, DR performs a hidden surface removal (HSR) phase. During the HSR phase, all the tile primitives are first rasterized only for position and depth, and the resultant fragments are Early-depth tested to setup the Depth buffer. Once HSR is complete, the second pass processes the tile primitives as usual along the raster pipeline (they are read, rasterized and depth-tested again), except that this time the Early-depth test performs optimal occlusion culling. Although the details of this technique in commercial systems are not fully disclosed, we have modeled in our framework an efficient implementation of it at the microarchitecture level, which is described in Subsection 4.3.1. In contrast to Z-prepass, DR does not perform the geometry processing twice. However, DR still has a non negligible cost: either it introduces a barrier in the graphics pipeline because the Fragment Processing stage is not started until HSR has completely finished for a tile, or extra hardware is required to perform the HSR of a tile in parallel with the rendering of other tile. Further details are given in Section 4.3.1.

Deering et al. [132] proposed a Deferred Shading architecture where the screen space is divided into bins indexed by scan line. First, all the geometry of a given frame is processed, sorted into bins, and stored in an intermediate buffer (Y-buffer). Then, when all the triangles of a given frame

have been processed, the following stages of the graphics pipeline are performed in two pipelines in sequence, the Triangle Processor Pipe (that rasterizes triangles creating fragments) and the Normal Vector Shader pipe (that computes the color of the fragments). The authors included multiple Triangle Processor and Normal Vector Shader pipes, providing a highly parallel architecture that was able to overcome the main memory access bottleneck. Later, Saito et al. [184] proposed a Deferred Shading scheme that avoids to compute the color of hidden surfaces. This technique proposes a two-pass scheme. In the first pass, the per-fragment data (depth, normals, texture coordinates, ...) is computed and stored in intermediate buffers referred as G-buffers. Then, the second rendering pass performs the fragment shading to compute the color of the visible fragments.

Ragan-Kelley et al. [179] propose a technique that decouples shading from visibility determination and allows to shade at a lower rate but yet enabling super-sampling effects. Rather than performing two rendering passes and determining the visibility in the first one, the authors propose a single pass technique that employs a memoization scheme, which caches shading results. If a fragment passes the visibility test, instead of computing a shading sample (color), the memoization buffer is checked for previous cached values at the same fragment's position. If the memoization buffer contains the value, it is reused avoiding the shading. Otherwise, the shading sample is computed. Based on the work of Ragan-Kelley et al. [179], Liktor et al. [164] propose to introduce Compact G-buffers (CG-buffer), that reduce the memory footprint of the G-buffer by means of reusing data of the G-buffer. Instead of storing data per every visible fragment that will be later processed in the Fragment Shading stage, the CG-buffer includes a visibility reference buffer, which holds references (per every visible fragment) that point to entries with shader inputs. Unlike the G-buffer, the CG-buffer avoids to store redundant entries that correspond to the same results shader output. Hence, this approach eliminates redundancy before fragments shading by reusing G-buffer results, rather than reusing fragment shading results like the previous approach.

Clarberg et al. [124] propose a Deferred Rendering method that is executed in two phases. The first one performs vertex processing (only position), rasterizes the geometry of the scene and determines the visibility. The second phase sorts the geometry into tiles, and then for each tile it repeats the vertex processing but only for the visible primitives, which allows to perform vertex processing of attributes, as well as attribute interpolation and fragment processing just for visible fragments. The authors only report bandwidth and fragment executions but no energy numbers.

Other Related Work

Arnau et al. [110] propose Parallel Frame Rendering (PFR), which exploits frame to frame coherency and temporal locality to improve the energy-efficiency of the GPU. The authors take advantage of the similarity between the texture data sets between consecutive frames, and propose a scheme that splits the GPU resources to process multiple consecutive frames in parallel (synchronizing the memory accesses of such frames), which increments the hit rate of the texture caches. Hence, the main memory bandwidth related to textures is reduced providing significant bandwidth and energy savings, as well as significant speed-up. In a later work, Arnau et al. [112] extend PFR in order to reduce overshading. Like with PFR, they render several frames in parallel, but they additionally introduce a memoization scheme that caches results of the Fragment Processors to avoid redundant executions. Once a fragment passes the visibility test, the memoization scheme

produces a signature of the inputs of the Fragment Processor for that fragment, and checks if it is present in the memoization buffer. If the signature is already present in the buffer, the scheme reuses a previous fragment shading result, which reduces overshading. If the signature is not found, the color of the fragment is computed in the Fragment Processors, and the corresponding result along with the corresponding signature are cached in the memoization buffer.

The unit of execution of the Fragment Processing is usually the quad fragment, a group of 2x2 fragments located at a 2x2 pixel region of the viewport (screen). Given a primitive, it may happen that not all the fragments in a group of 2x2 are covered by a primitive. Hence, when the processor performs the fragment shading for not fully covered quad fragments its resources are underutilized. This problem is exacerbated in the presence of micro-triangles. Fatahalian et al. [137] propose a scheme to reduce overshading caused by micro-triangles by means of gathering and merging partially covered quad fragments, that are generated from adjacent tessellated primitives, into a single quad fragment before doing fragment shading.

Sathe et al. [186] propose a scheme that aims at reducing overshading for micro, medium, and large sized triangles when using Multi-sample anti-aliasing (MSAA). MSAA is a technique that, given a fragment, uses four sampling points for color and depth, while fragment shading is still performed just once per fragment. MSAA improves the appearance of the frame in high frequency zones (i.e. object boundaries). On the contrary, the quality of the frame in low-frequency zones hardly improves in the presence of MSAA. The authors exploit that fact and propose a hardware unit that for a group of connected primitives detects when a primitive does not cover the center of a given pixel (but cover at least one of its four sampling points) and skips the shading for that triangle. Instead, the scheme reuses the shading result of the connected primitive that indeed covers the center of the pixel.

1.3.2 Collision Detection

There are many proposals to detect object interference through geometric computations, for example by computing intersections between pairs of bounding volumes (spheres, capsules, boxes, k-DOPS, ...), or by computing the distance between points in the space. CD algorithms are intrinsically quadratic with respect to the number of objects and their surfaces. To alleviate this cost CD is often split into two phases:

- **Broad Phase:** Fast and simple, and often coarse grain, tests applied to pairs of collisionable objects in a scene. In this phase it is common to employ some kind of spatial partitioning in order to reduce the number of pairs of objects to be tested. The pairs of objects whose bounding volumes collide are added to the Potentially Colliding Set (PCS), whose elements are tested in the narrow phase.
- **Narrow Phase:** Slow and complex test that employs more accurate shapes than the broad phase to test the collision for the pairs of objects that were suspect to collide in the previous phase.

There exists a large body of research on CD [154, 135, 158, 196]. Both broad and narrow phases

can be executed in a CPU or a GPGPU, depending on the characteristics of the specific platform. Broad phase algorithms are simple to parallelize, whereas narrow phase algorithms are usually, for a given pair of objects, control-intensive. In most cases, the narrow phase of CD is executed in the CPU because of the non-regular nature of the computations, and in low-power systems the broad phase is executed in the CPU as well.

There are versions of some CD algorithms written for GPGPU. Even though the innermost loops of the narrow phase algorithms are difficult to parallelize because of its control-intensiveness, these algorithms can exploit parallelism by evaluating the narrow phase for multiple colliding elements in parallel. However, although they can perform the computation fast, GPGPU schemes must still bring the geometry of the colliding elements from CPU memory to GPU. The cost of transferring this geometry from memory is not negligible and in some cases it could represent more time than the computational cost itself. Lee et al. [161] take into account these costs and report a 14x speedup of a very accurate CD algorithm (GJK [140]) on a GPGPU compared to the CPU version.

Many proposals apply a spatial hierarchical partitioning of the objects in a scene. Ar et al. [107] use BSP-trees, which subdivide the space in two sub-spaces that satisfy a given requirement. In early computer graphics systems the BSP-trees were used along with the Painter's algorithm to render scenes in the absence of the Depth buffer algorithm. Fan et al. [136] employ octrees, a tree where every node has exactly eight children nodes that represent one eighth of the space represented by the parent node. Wang and Liu [198] employ AABB-trees, a tree composed of AABBs (Axis Aligned Bounding Boxes) where the root of the tree is an AABB that contains all the geometry of the scene and every child of a node is an AABB bounded by the AABB of the parent node. Lawlor et al. [160] propose to subdivide the scene in an axis-aligned grid of voxels, where every voxel contains a list with the objects that it overlaps. Then, any other CD scheme is used to detect the collision points for every voxel's list.

He et al. [149] propose a scheme that creates Bounding Volume Hierarchies (BVHs) based on dynamic clustering of the objects in a scene. First of all, the objects are decomposed in non self-colliding leaf-clusters. Then, an intermediate step merges such clusters and creates intermediate-clusters that maintain the non self-colliding property. The merge step creates a binary tree where the last-level nodes are leaf-clusters, and the other nodes are intermediate-clusters. For all the intermediate-clusters the scheme computes appropriate bounding volumes. A BVH is created starting from the parents of the leaf-clusters up to the root clusters of the binary tree. Then, the scheme performs overlap tests between the nodes of the tree. If a given overlap test reaches the leaf level of the binary tree, a BVH is computed for every leaf-cluster involved in the test. The process of CD is then resolved by testing the collision of the corresponding BVHs of the leaf-clusters involved in the possible collision. When testing the BVHs of two leaf-clusters, if the last level of the hierarchy is reached, the scheme uses triangle tests to check the collisions.

Du et al. [134] present a parallel Continuous CD (CCD) algorithm that distributes the work across a high-performance shared-memory system that includes both GPUs and CPUs. For every object in a scene, the scheme creates a Sphere Bounding Volume (SBV) and computes the trajectories of the SBVs performed between two time steps. Such trajectories are projected on to a Cartesian coordinate system. Then, a classical Sweep and Prune [29] algorithm is performed in the GPU in order to discard non overlapping trajectories and create the PCS. For a given pair of objects in the PCS, the scheme creates a BVH for each object. Finally, the nodes of the BVH are tested to collide.

To detect the collisions at primitive level granularity the scheme uses an interval-iteration method.

Image-Based CD

There is a group of CD algorithms known as Image-Based CD (IBCD). IBCD consists in the rasterization of the surfaces of the scene objects and the detection of their intersections based on the pixel depths of the corresponding fragments [115]. These kind of techniques have been proposed to exploit the computing power of graphics processors and their ability to rasterize polygons efficiently. Shinya et al. [189] and Rossignac et al. [183] opened the path to IBCD with their pioneering work.

Myszkowsky et al. [170] propose a technique based on the Depth buffer and on checking the changes in the Stencil buffer between a frame and the following one, and suggest including a hardware assisted polygon sorting in order to dramatically increase the performance of the technique.

Baciu and Wong [113] propose a detection scheme based on reducing the region of the Stencil buffer to be tested, which reduces the main memory traffic and thus minimizes the need for the specific hardware pointed out in the previous work [170].

Baciu and Wong [114] and Heidelberger et al. [151] show hybrid techniques using both object space and image-based CD. They sometimes use more than one rendering pass, since they rely on a rasterization system with only one depth value per pixel in the Depth buffer at any given time. The latter work performs the collision detection on the CPU. They expand this work to handle auto-collisions with deformable geometry [152]; however, both works rely on reading back to the CPU the results of the rasterization (Depth and Stencil buffers).

Knott and Pai [157] make use of the Color, Depth and Stencil buffer and several rendering passes in order to detect collisions among all the objects to be tested. CULLIDE [142], proposed by Govindaraju et al., can reduce the aforementioned buffer read-back overhead by making use of occlusion queries. In a later work [143], the authors improve the accuracy of the method by computing conservative overlap tests between the primitives and avoiding-to-miss collisions due to viewport resolution. Faure et al. [138] propose to detect collisions between two 3D objects using surface rasterization in three orthogonal directions to discard objects that do not overlap in one of the axes. Then, they perform the narrow phase of the CD in the CPU.

Chen et al. [121] also employ a classic broad and narrow phase scheme. This scheme executes the broad phase in the CPU, which uses AABBs to rapidly discard non-overlapping objects. Additionally, in the broad phase they compute the region where the two AABBs overlap, commonly known as Region of Interest (ROI). Then, the volume of the resultant pair of objects that is inside the ROI is voxelized in real-time using the GPU. The voxels of each object are stored in an independent 2D texture that describes the volume of the object. Finally, the collision test is performed in the GPU by pair-wise comparing every voxel of the 2D textures. If at least there is one common voxel, the pair of objects is reported to collide.

1.4 Thesis Overview and Contributions

The goal of this thesis is to propose novel and effective techniques to eliminate redundant computations performed in real-time computer graphics applications with special focus on mobile GPU micro-architecture. Improving the energy-efficiency of CPU/GPU systems, which are extensively deployed in mobile devices, is key to enlarge their battery life. Our main contributions are Visibility Rendering Order and Render-based Collision Detection. The former is a technique that reuses visibility information of previous frames to eliminate redundant computations of hidden surfaces in the Fragment Processors of the GPU for the current frame being rendered. The latter one is a technique that avoids redundant computations related to CPU Collision Detection by reusing intermediate rendering results employed to perform accurate CD in the GPU. These works are implemented on top of a conventional mobile GPU being suitable for both Tile Based and Immediate-Mode GPU architectures. Furthermore, we also propose a micro-architectural implementation of a Tile Based Deferred Rendering GPU and we make a performance and energy evaluation of the Z-prepass software technique. Below we present the problems we deal with and the approaches we take to solve them, as well as providing a comparison with related work.

1.4.1 Visibility Rendering Order

Figure 1.7 introduces a simplified conventional graphics pipeline. The GPU receives vertices and processes them in the Geometry Pipeline, which generates triangles. These are then discretized by the Rasterizer, which generates fragments that correspond to pixel screen positions. Then, fragments are sent to the Fragment Processing stage, which performs the required texturing, lighting and other computations to determine their final color. Finally, the Depth test compares each fragment's depth against that already stored in the Depth Buffer to determine if the fragment is in front of all previous fragments at the same pixel position.

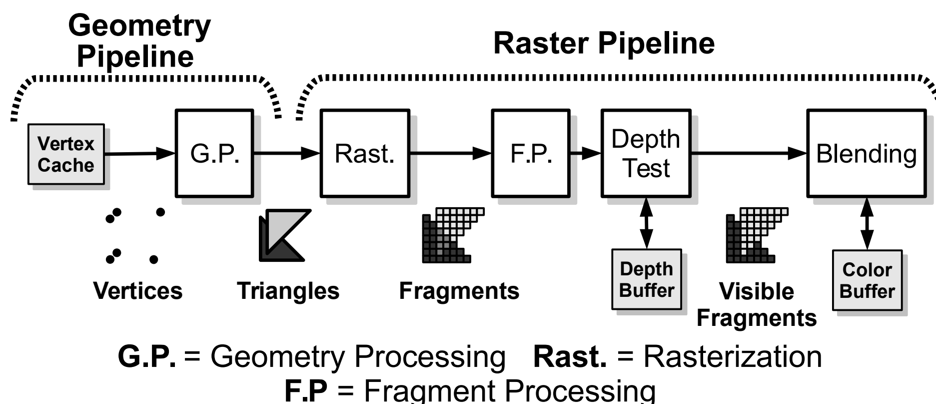


Figure 1.7: Simplified version of the Graphics Pipeline.

When the GPU realizes that an object or part of it is not going to be visible, all activity required to compute its color has already been performed, with the consequent waste of time and energy, especially in the Fragment Processing stage, which is the most power consuming stage of the graphics pipeline [176]. To help discard occluded surfaces earlier in the pipeline, most current

GPUs include an Early-Depth test before the fragment processing stage.

In first place, we evaluate the amount of overshading in mobile graphics workloads and the effectiveness of Early-depth test to reduce it. Early-depth reduces an important part of the overshading but there is significant room for improvement because, to be effective in avoiding the rendering of hidden surfaces, it requires opaque geometry to be processed in a front-to-back order. We also evaluate a Deferred Rendering scheme, which obtains perfect occlusion culling at fragment level. However it includes an extra render pass to determine visibility, which has a non negligible cost.

In second place, we propose a novel architectural technique for mobile GPUs, Visibility Rendering Order (VRO), which reorders objects front-to-back entirely in hardware to maximize the culling effectiveness of the Early-depth test and minimize overshading, hence reducing execution time and energy consumption. VRO exploits the fact that the objects in graphics animated applications tend to keep their relative depth order across consecutive frames (temporal coherence) to provide the feeling of smooth transition. VRO keeps visibility information of a frame, and uses it to reorder the objects of the following frame. Since depth-order relationships among objects are already tested by the Depth Test, VRO incurs minimal energy overheads. It just requires adding a small hardware to capture the visibility information and use it later to guide the rendering of the following frame. Moreover, VRO works in parallel with the graphics pipeline, so negligible performance overheads are incurred. We illustrate the benefits of VRO using various unmodified commercial 3D applications for which VRO achieves 27% speed-up and 14.8% energy reduction on average over a state-of-the-art mobile GPU. This work is included in a paper that is currently in reviewing process:

- "Visibility Rendering Order: Improving Energy Efficiency on Mobile GPUs through Frame Coherence".
  Enrique de Lucas, Pedro Marcuello, Joan-Manuel Parcerisa, and Antonio González.
  To be published.

Multiple works have analyzed how overshading can be reduced by culling the geometry at primitive level granularity through the use of occlusion queries [118, 187]. When using occlusion queries, the application usually sends a query with a Bounding Volume of the object to the GPU to be rasterized and depth-tested. Eventually, the result of the query is sent to the driver and if the Bounding Volume was occluded, the application will not send the object to the GPU. Given that at some point the application must know the result of the queries, they may introduce CPU stalls and produce GPU starvation. Furthermore, some drivers let the GPU render several frames behind the CPU by actually queuing the rendering commands [130, 92, 90], which exacerbates these problems. Like the Early-depth, occlusion queries require to sort the queries (and the objects) front-to-back to perform well. On the contrary, VRO does not suffer by these limitations. On the one hand VRO is fully integrated into the GPU, so the application does not need to receive any feedback from VRO. On the other hand, VRO reuses the results produced in the Depth test of the actual rendering commands of the application, instead of introducing extra work (occlusion queries) to reduce overshading.

Govindaraju et al. [141] sort the primitives of every object of a scene in a front-to-back order from a given viewpoint. They assume that the objects do not overlap, so the scheme only avoids

intra-object overshading. Furthermore, they are not able to handle cycles in the sorting process whereas VRO is able to produce a Visibility Rendering Order in the presence of cycles, which are highly common on 3D scenes. There are other approaches focused on reducing intra-object overshading. Nehab et al. [171] and Sander et al. [185], propose a pre-processing scheme that sorts the triangles in a view-independent order to reduce overdraw. However, they focus on static meshes and produce a single order per mesh. On the other hand, Han et al. [146] target animations and produce different view-dependent orders per object, which are used by the application depending on the orientation of the objects with respect to the camera. These techniques, focused on reducing intra-object overshading are complementary to VRO, which reduces inter-object overshading.

Rendering the objects in a front-to-back order effectively reduces overshading but unfortunately it is not the general case in commercial applications. Our proposal creates a view-dependent front-to-back order that effectively reduces overshading in a transparent manner to the programmer. VRO does not require neither extra Vertex Processing, Rasterization nor Early-depth executions. Furthermore, VRO can handle both static and animated scenes and is able to create a rendering order of a scene even in the presence of cycles between different objects.

1.4.2 Render Based Collision Detection

Graphics animation applications such as 3D games represent a large percentage of downloaded applications for mobile devices and the trend is towards more complex and realistic scenes with accurate 3D physics simulations, like those in laptops and desktops. Collision detection (CD) is one of the main algorithms used in any physics kernel. CD is one of the most important algorithms since it identifies the contact points between the objects of a scene, and determines when they collide. However, real-time highly accurate CD is very expensive in terms of energy consumption and this parameter is of paramount importance for mobile devices since it has a direct effect on the autonomy of the system. This work proposes a novel energy-efficient high-fidelity CD scheme that leverages some intermediate results of the rendering pipeline to perform CD.

In first place, we give a brief introduction to CD, in particular to a group of algorithms known as Image-Based Collision CD (IBCD). These algorithms rely on the rasterization of the surfaces of the scene objects and the detection of their intersections based on the pixel depths of the rasterized fragments [115]. These kind of techniques have been proposed to exploit the computing power of graphics processors and their ability to rasterize polygons efficiently.

In second place, we introduce our proposal: Render Based Collision Detection (RBCD), which belongs to this family of techniques. RBCD is based on the observation that most of the tasks required for IBCD (e.g., vertex processing, projection, rasterization, etc.) are also performed during image rendering. Hence, we propose to integrate CD and image rendering within the GPU pipeline hardware, so that redundant tasks are done just once. With minor hardware extensions and minor software adaptations our technique reutilizes some intermediate results of the rendering pipeline to perform the CD task. Some of these adaptations include allowing the software to pass collisionable object identifiers to the GPU, selectively deferring face culling, and adding small, specific hardware to detect face intersections based on per-fragment location and depth.

In third place, we show the benefits of RBCD in a CPU/GPU system. Comparing RBCD with

a conventional CD completely executed in the CPU, we show that its execution time is reduced by almost three orders of magnitude (600x speedup), because most of the CD task of our model comes for free by reusing the image rendering intermediate results. Although not necessarily, such a dramatic time improvement may result in better frames per second if physics simulation is in the critical path. However, the most important advantage of our technique is the enormous energy savings that result from eliminating a long and costly CPU computation and converting it into a few simple operations executed by a specialized hardware within the GPU. Our results show that the energy consumed by CD is reduced on average by a factor of 448x (i.e., by 99.8%). These dramatic benefits are accompanied by a higher fidelity CD analysis (i.e., with finer granularity), which improves the quality and realism of the application. This work has been published in the proceedings of 48th International Symposium on Microarchitecture (MICRO):

- "Ultra-low Power Render-Based Collision Detection for CPU/GPU Systems". Enrique de Lucas, Pedro Marcuello, Joan-Manuel Parcerisa, and Antonio González. International Symposium on Microarchitecture, 2015.

Myszkowsky et al. [170] propose a technique based on the Z-buffer and on checking the changes in the Stencil buffer between a frame and the following one, and suggest including a hardware assisted polygon sorting in order to dramatically increase the performance of the technique. Baciu and Wong [113] propose a detection scheme based on reducing the region of the Stencil buffer to be tested, which reduces the main memory traffic and thus minimizes the need for the specific hardware pointed out in the previous work [170]. Baciu and Wong [114], and Heidelberger et al. [151] show hybrid techniques using both object space and image-based CD. They sometimes use more than one rendering pass, since they rely on a rasterization system with only one depth value per pixel in the z-buffer at any given time. The latter work performs the collision detection on the CPU. This work is expanded [152] to handle auto-collisions with deformable geometry; however, both works rely on reading back to the CPU the results of the rasterization (Depth/Stencil buffer). Knott and Pai [157] make use of the Depth/Stencil and Color buffer plus several rendering passes in order to detect collisions among all the objects to be tested. CULLIDE [142], proposed by Govindaraju et al., can reduce the aforementioned buffer read-back overhead by making use of occlusion queries. In a later work [143], the authors improve the accuracy of the method by computing conservative overlap tests between the primitives and avoiding-to-miss collisions due to viewport resolution. Faure et al. [138] propose to detect collisions between two 3D objects using surface rasterization in three orthogonal directions, and performing the CD in the CPU. Unlike previous works, RBCD requires neither multiple rendering passes nor reading back the results of the rasterization. Furthermore, our scheme is projection independent. RBCD performs the CD of the scene in the RBCD unit added to the GPU and then reports the detected contact points between collisionable objects to the CPU.

Gilbert et al. [140] propose Gilbert-Johnson-Keerthi (GJK), an algorithm that computes the minimum distance between two convex sets of vertices. Lee et al. [161] report a 14x speedup of GJK on a GPGPU, whereas we are reporting a speedup of 3400x with our technique. Despite GPGPU schemes can perform the computation fast, they must still bring the geometry of the colliding elements from main memory to the GPU, whereas our proposal takes advantage of the information that has already been generated in the GPU to render the image. The cost of transferring this geometry from memory is not negligible and in some cases it could represent more time than the

computational cost itself. This transfer time as well as the corresponding energy is saved by our scheme. Additionally, our scheme reuses most of the computations from the rendering performed in the GPU, which provides further energy savings.

1.4.3 Other contributions

Micro-architectural evaluation of Deferred Rendering

Deferred Rendering (DR) is a hardware technique that avoids overshading through computing the Depth Buffer before starting fragment shading. We perform a micro-architectural evaluation of two different approaches to perform Deferred Rendering: sequential DR and parallel DR. Sequential DR is a naïve implementation that stalls the rest of the Raster Pipeline while performing HSR. The sequential implementation badly hurts both performance and energy compared with the baseline GPU. For this scheme, the execution time increases for every one of the benchmarks tested, being the slowdown 23% on average. Regarding energy consumption, it increases around 6% on average when compared with the baseline GPU. These huge overheads are due to the fact that the total time of the HSR stage (only depth rasterization plus depth test) greatly exceeds the savings provided by the overshading reduction. Nevertheless, these huge overheads can be removed by performing the HSR stage in parallel with the other stages of the Raster Pipeline. Thus, in this optimized scheme (parallel DR), while the HSR is being executed for tile *i+1*, the rest of the Raster Pipeline is executed in parallel to render the tile *i*. Even though this parallel implementation of DR introduces a non negligible amount of extra hardware (6% area overhead w.r.t baseline GPU), it outperforms sequential DR in both performance and energy.

Evaluation of software Z pre-pass

Z pre-pass is a common software approach aimed at reducing overshading that has gained pace in the last years and it can be considered the standard on IMR and TBR GPUs to reduce overshading in complex games with multiple dynamic objects and costly Fragment Shaders. Some vendors like ARM recommend to use it when setting the rendering order of the objects is not possible because of the complexity of the scene [4]. Z pre-pass exploits the Early-depth test by means of two rendering passes. The first pass performs pipeline stages up to the Early-deph test, which stores the depths of the visible fragments in the Depth Buffer. In the second pass, the full pipeline is executed and only the visible fragments pass the Early-depth, so only visible fragments are executed in the Fragment Processors. This technique doubles the cost of some stages of the pipeline, which is unacceptable in many scenarios. Take into account that even if the benefits of Z pre-pass are greater than its cost, the extra depth pass represents a large portion of the total rendering time. For example, Z pre-pass can be found on The Blacksmith [6], which is a cutting edge real-time demo of the last version of Unity [21] made to show the most advanced graphics features that the game engine offers. We have studied three different frames of Blacksmith, where Z pre-pass represents 41.4%, 29.3% and 26.1% of the total rendering time respectively.[1] In common mobile graphics workloads included in our set of benchmarks Z pre-pass produces a slowdown in

[1] Real hardware measures reported by Renderdoc [20] using an NVIDIA GTX 950 GPU .

all the cases (0.82x on average). Z pre-pass also increases the total energy consumption, 1.28x on average. The increase in the main memory traffic and the extra execution time greatly penalize the energy consumption. According to these results, Z pre-pass would not be a suitable technique to reduce neither execution time nor energy consumption on applications targeted for low power GPU because the overhead incurred by the extra rendering pass more than offsets the time and energy savings of the smaller fragment processing. Nevertheless, it makes sense to apply Z pre-pass in a context with higher computational cost per fragment processed (more details in Section A.3.1).

Immediate Mode Rendering VRO

We have evaluated VRO over an IMR GPU architecture. Like VRO, IMR-VRO includes a small hardware unit that stores the order relations among the objects of a scene of the current frame in a buffer. This information is used in the next frame to create a Visibility Rendering Order that this time guides the Command Processor of the GPU. IMR-VRO outperforms state-of-the-art techniques in performance and energy consumption by reducing the overshading without the need of an expensive extra rendering pass to determine visibility. IMR-VRO achieves up to 1.32x speed-up (1.16x on average) and down to 0.79x energy consumption (0.85x on average) with respect to the baseline IMR GPU. IMR-VRO is much more efficient than Z-prepass because most of the computations required to create the Visibility Rendering Order are reused from the normal rendering, while Z-prepass requires an extra renderization pass that introduces significant overheads.

2

Background

2.1 Graphics Rendering Pipeline

This chapter presents an overview of different concepts relative to graphics rendering. It is presented the classic scheme of the graphics rendering pipeline including its different stages. Then, we present a description of the microarchitecture for mobile GPUs. The purpose of this chapter is not only to focus the context of this thesis in its technical aspects, but also to introduce the terminology we will use through the document. Rather than making an exhaustive review of computer graphics, the intention of this section is to briefly introduce the graphics rendering pipeline. For more detailed information the reader has available excellent books that cover multiple aspects of computer graphics [104, 166].

The graphics rendering pipeline, also commonly referred to as the graphics pipeline or just the rendering pipeline, is an abstract model whose duty is to render 2D images (frames) from descriptions of geometric models and other information like the point of view, light sources and more. It is usually divided in three conceptual stages (see Figure 2.1): Application, Geometry and Rasterization.

2.1.1 Application Stage

The application stage is composed of the software operations required to prepare the geometric models and all the associated information that is necessary to render them in the following stages of the pipeline. This stage receives user inputs, computes the corresponding reactions, updates the state of the application accordingly, and finally sends the geometry to the next stage of the pipeline. This group of tasks may be referred to as a render step. Since the application stage is a

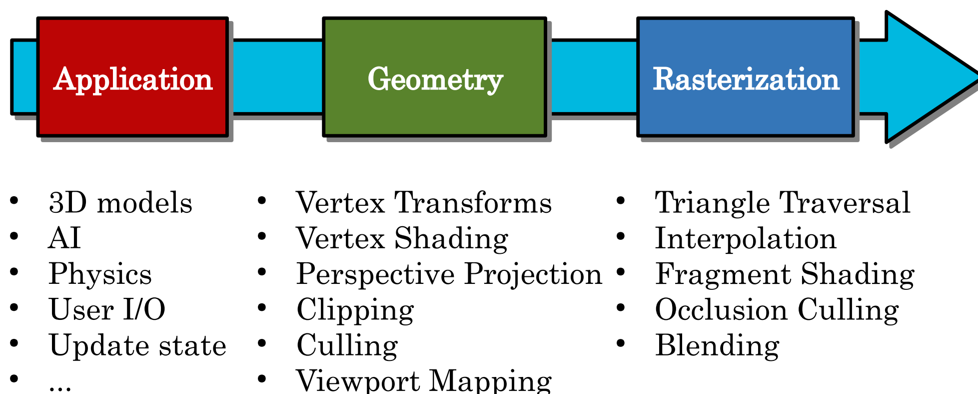


Figure 2.1: Conceptual stages of the Graphics Rendering Pipeline.

software stage, the developers have the freedom to implement the render step which they consider appropriate. However, software APIs like game engines or GUI frameworks usually offer their own implementation of the render step [181, 127], which is an intermediary between the application and the graphics API.

A process commonly included in the application stage is the collision detection, which is part of the physics simulation. It provides realistic physical behavior to the visual objects of the application, whose velocity and direction must be affected by collisions with other objects.

2.1.2 Geometry Stage

The geometry stage performs all vertex-level (vertex transformations, vertex shading) and almost all polygon-level (primitive assembly, clipping, face culling) operations of the graphics pipeline.

Vertex Transformations

Figure 2.2 shows the different vertex transformations applied in the graphics pipeline, which consist on the conversion of vertex coordinates from one coordinate system to another by means of multiplying a transformation matrix with the position of the vertices. Besides the transformation between coordinate systems, operations like translation, rotation and scaling may also be applied to the vertices. There are several coordinate systems in the graphics pipeline:

- **Object Coordinates:** Initially each object is created with its vertices relative to its own coordinate system. This is the coordinate system defined by the creator of the object in order to model the object by positioning every vertex in the right place regarding to the other vertices of the model.

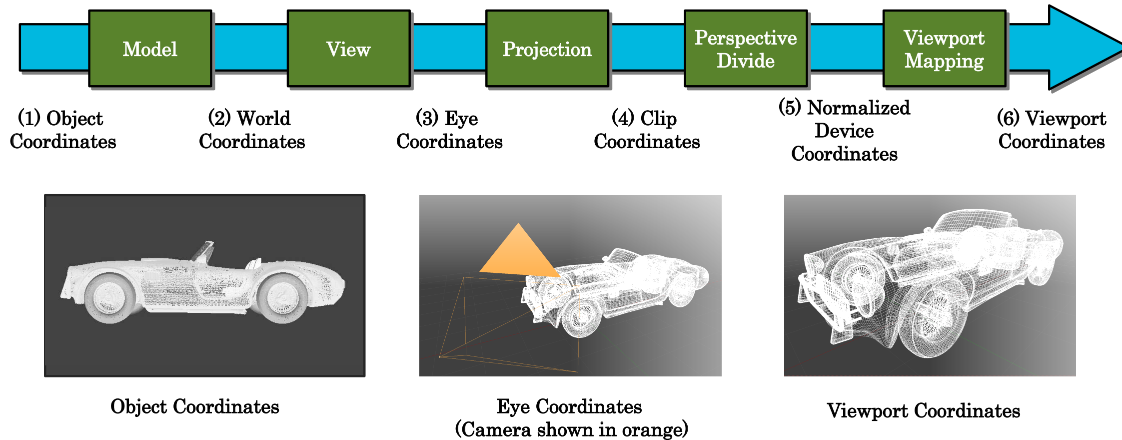


Figure 2.2: Vertex-level transformations in the Graphics Rendering Pipeline. (1) The vertices of the 3d model are in Object Coordinates. (2) The vertices are scaled, rotated and translated to World Coordinates. (3) The vertices are positioned in the camera scope transforming them to Eye Coordinates (see detail of camera in orange outline). (4) The vertices transformed to Clip Coordinates by projecting them onto the near clip plane. (5) Perspective correction is applied to transform vertices to Normalized Device Coordinates. (6) Viewport transform is applied to translate vertices to Viewport Coordinates. Car 3D model courtesy of Alexander Bruckner [24].

- **World Coordinates:** This is the coordinate system where all the objects are positioned (translated, rotated, scaled) after the model transform.
- **Eye Coordinates:** Also called camera coordinates or viewpoint coordinates, this is the coordinate system where the camera is statically located at the origin, looking down the z-axis.
- **Clip Coordinates:** Also called projection coordinate system, this coordinate system defines how the vertices data are projected onto the screen. It also defines the volume of the world where objects may be visible from the camera's point of view: the view-frustum. A vertex (X, Y, Z, W) is outside the view-frustum if any of its coordinates X, Y, or Z is outside the range (-W, W). As Figure 2.3 shows, this volume may be a frustum (perspective projection) or rectilinear (orthogonal projection).
- **Normalized Device Coordinates (NDC):** This coordinate system is used to measure relative positions on the screen, but it has not been transformed to screen pixels yet. It is not achieved by applying a matrix transform to the vertex (X, Y, Z, W), but by dividing the first three vertex coordinates X, Y and Z by W -(X/W, Y/W, Z/W)-. Its values are normalized and range from -1 to 1 in all three axes in OpenGL (see Figure 2.4), while for DirectX the z-axis range is (0, 1).
- **Viewport Coordinates:** Also commonly referred to as window or screen coordinates. The vertices in NDC coordinates are translated and scaled to be positioned in terms of a viewport with rendering resolution *Width* x *Height*. This coordinate system covers the 2D plane that is actually displayed on the screen, where (0, 0) is the bottom left and (*Width - 1*, *Height* - 1) is the top right of the plane. For example, for rendering resolution 1280 x 720 the geometry that

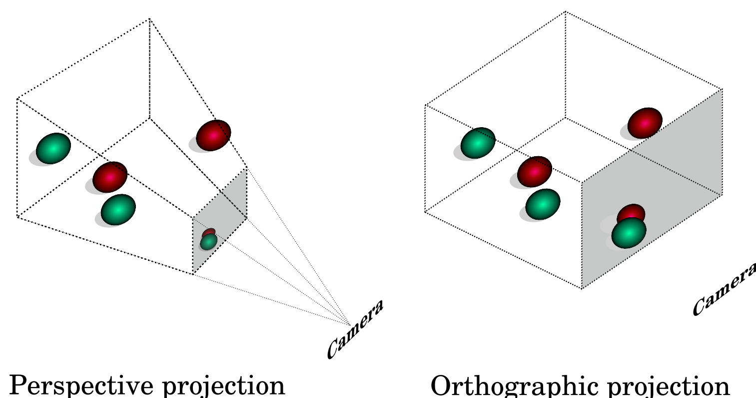


Figure 2.3: Perspective (left) and orthographic (right) viewing volumes.

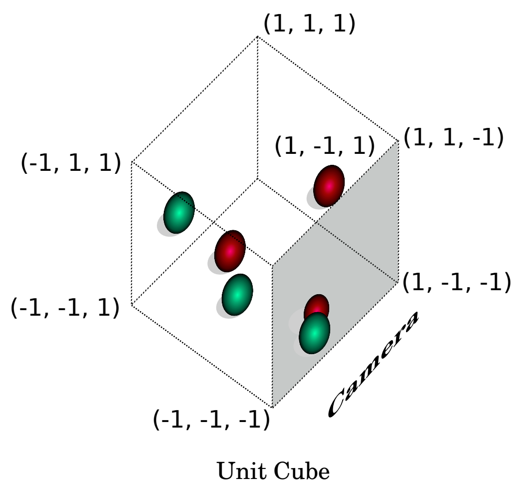


Figure 2.4: OpenGL normalized viewing volume.

was inside normalized viewing volume is placed between (0, 0) and (1279, 719) coordinates. The geometry in viewport coordinates is passed to the rasterization stage to become fragments.

Vertex Shading

In order to render realistic scenes not only the position of the objects is transformed across several coordinate systems, but also it is necessary to compute a set of attributes per every vertex of the object. From simple colors for every vertex to complex elaborations that include texture coordinates, normals, and light sources among others, the process of determining these vertex attributes is commonly known as vertex shading. In the vast majority of current systems, these per vertex computations, as well as the transformations applied to the vertex positions are included in the vertex shading stage, which is executed in programmable hardware.

Primitive Assembly

Given the input stream of vertices and the connectivity among them (topology), the primitive assembly stage creates simple polygons named primitives, sometimes also referred to as base primitives (points, lines and triangles). Figure 2.5 shows some examples of topology. As can be seen, the selection of the topology is essential to optimally describe the geometry. For example, in order to describe two adjacent triangles using GL_TRIANGLES, we need six vertices. However, if we use GL_TRIANGLE_STRIP four vertices are enough. Primitive Assembly is an essential task that not only groups vertices into primitives, but also allows to apply optimizations at primitive level like Clipping and Culling.

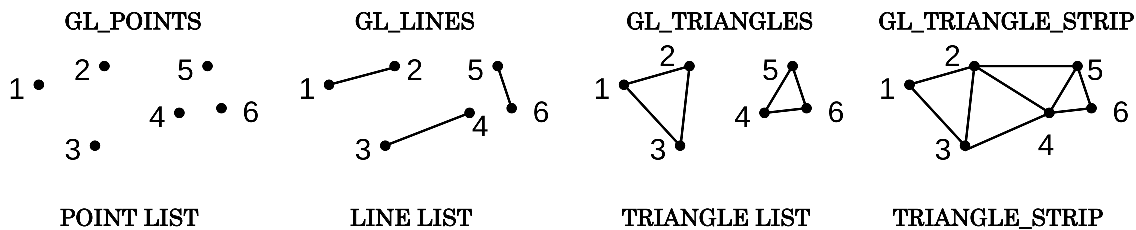


Figure 2.5: Examples of common topologies used both by OpenGL and DirectX [167].

- **Clipping:** Once a primitive is in Clip Coordinates we know if it lies inside or outside the view-frustum. Every vertex (X, Y, Z, W) of a primitive is clipped by comparing (X, Y, Z) with W. If all three components X, Y and Z are in the range (-W, +W), the vertex is inside the view-frustum. Otherwise the vertex is outside the view-frustum and must be clipped. Given a primitive, there are three possible cases:

  1. Trivial accept: All the area of a primitive lies inside the view-frustum.
  2. Clip: The primitive is partially inside the view-frustum.
  3. Trivial reject: All the area of the primitive lies outside the view-frustum.

  Figure 2.6 shows the six planes that form the view-frustum (left, right, top, bottom, near and far). For simplicity, Figure 2.7 shows the basic cases of primitive clipping for triangles, lines and points against the top, left, right and bottom clipping planes that form the viewport. When a primitive is partially outside the view-frustum there may be created new primitives. However this is not necessary and the system may leave the primitive intact. In such case, the screen mapping stage will discard the fragments outside the view-frustum. Common algorithms to perform clipping are the Cohen-Sutherland [25] and the Sutherland-Hodgman [27] algorithms for lines and polygons respectively.

- **Culling:** Once the triangle primitives are in Normalize Device Coordinates, they have a particular orientation regarding the view-point (camera). For a given triangle with vertices *1*, *2* and *3*, the orientation is defined by how the vertices rotate: clock-wise or counter-clock-wise (see Figure 2.8). One of the winding modes is selected as front face, and the other as back face. Commonly, when the 3D model is created, the modeling program sets the rotation of

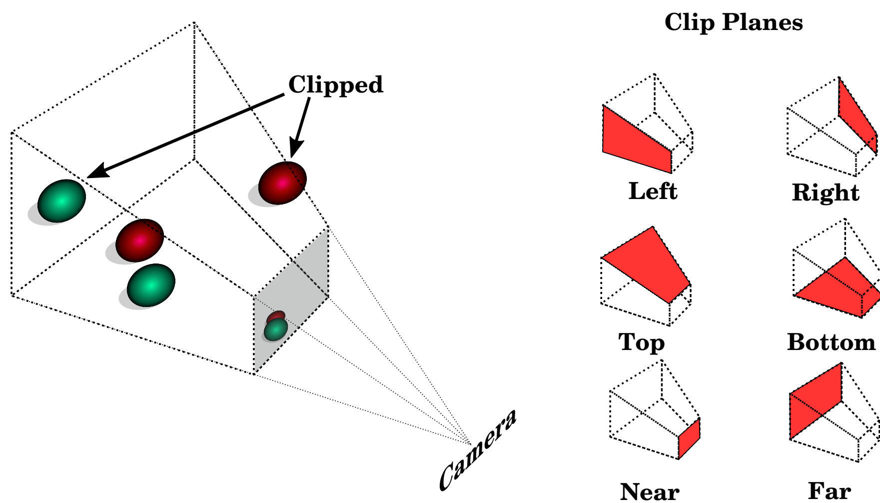


Figure 2.6: View-frustum including viewport.

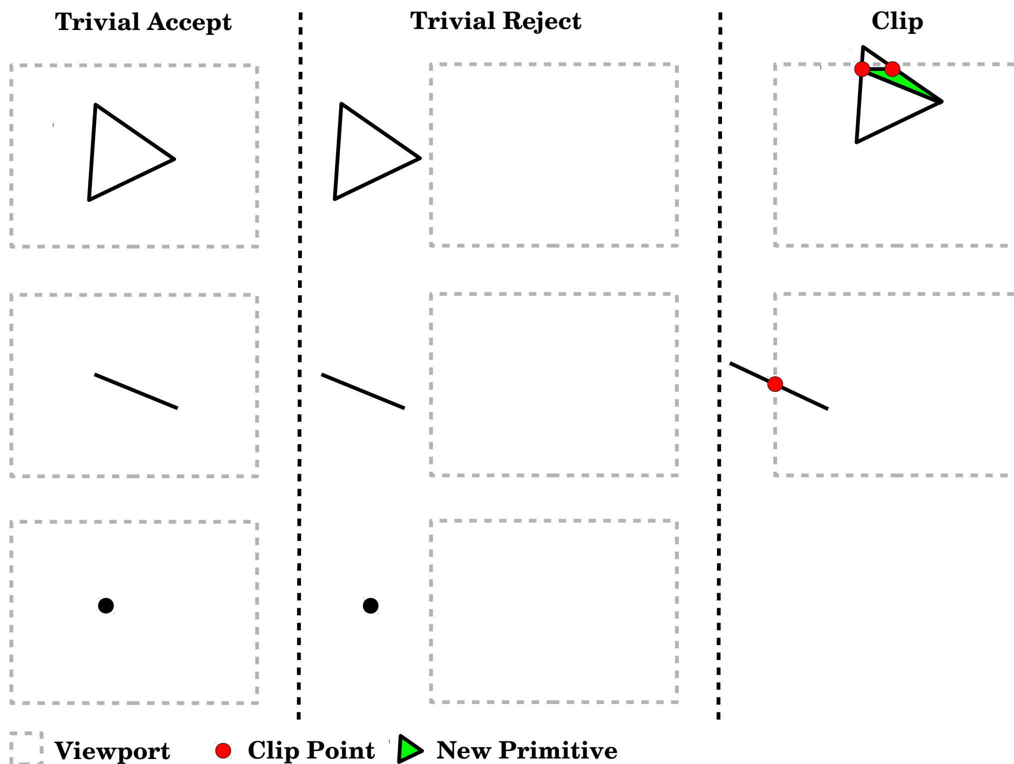


Figure 2.7: Clipping cases for triangles (top), lines (middle) and points (bottom).

the triangles that face to the camera (front-face) as counter-clock wise, and the triangles that do not face to the camera (back-face) as clock wise. Furthermore, with face culling the user

usually selects which triangles to cull: front-facing triangles, back-facing triangles (set by default), or both. Whenever the face culling stage receives a triangle, it tests the windowing mode of the triangle by performing a dot product that obtains the normal vector of the triangle. If the sign of the normal z-component is positive, the triangle is a front-face, otherwise the triangle is a back-face. Depending on the sign of the z-component of the triangle's normal and the selected culling mode the triangle is either culled or passes to the rasterization stage. Figure 2.9 shows the 3D model of a sphere (top-left), and the detail showing the triangles that pass the face culling test (top-right). Note the difference in the number of wires that are shown in the bottom-left image (face culling disabled) and in the bottom-right image (face culling enabled to cull back-faces).

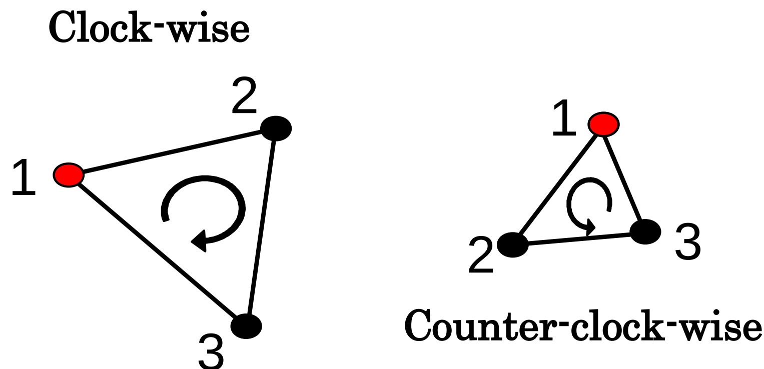


Figure 2.8: Clock wise and counter-clock wise winding triangles.

2.1.3 Rasterization

The rasterization stage is the third conceptual stage of the graphics pipeline. It receives the primitives of the objects in viewport coordinates (screen coordinates) and produces the color for the set of pixels covered by such primitives. As Figure shows 2.10, the rasterization stage performs five main tasks:

- **Triangle Traversal:** For every pixel covered by a primitive (which is already projected onto the screen plane) the rasterization stage creates a fragment, thus creating a discretized representation of the primitive. Every fragment corresponds to a pixel of the viewport. Scanline [83] and Edge Function [82] algorithms are common methods to perform triangle traversal, being Edge Function algorithm generally used today. This algorithm employs the following edge functions to determine the position of a point P with respect to the edges of a primitive defined by vertices $v0$, $v1$ and $v2$ (see Figure 2.11):

  1. $E01(P) = (P.x - v0.x) * (v1.y - v0.y) - (P.y - v0.y) * (v1.x - v0.x)$
  2. $E12(P) = (P.x - v1.x) * (v2.y - v1.y) - (P.y - v1.y) * (v2.x - v1.x)$
  3. $E20(P) = (P.x - v2.x) * (v0.y - v2.y) - (P.y - v2.y) * (v0.x - v2.x)$

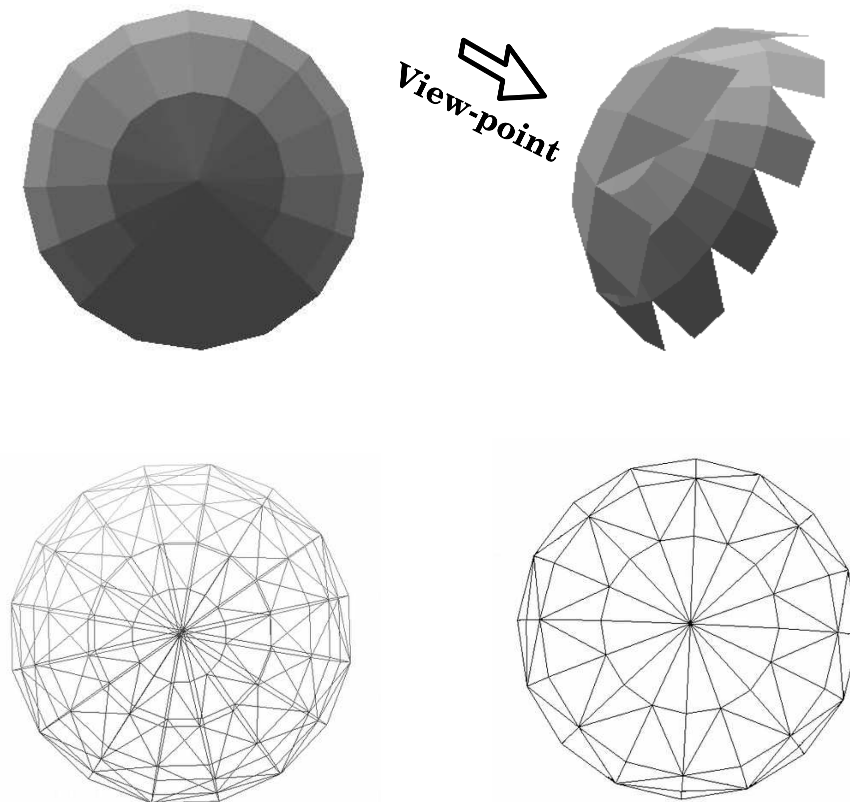


Figure 2.9: 3D model of a sphere (top-left), detail showing the effect of face culling (top-right), wire-frame view of the sphere without face culling (bottom-left), wire-frame of the sphere with face culling enabled (bottom-right).

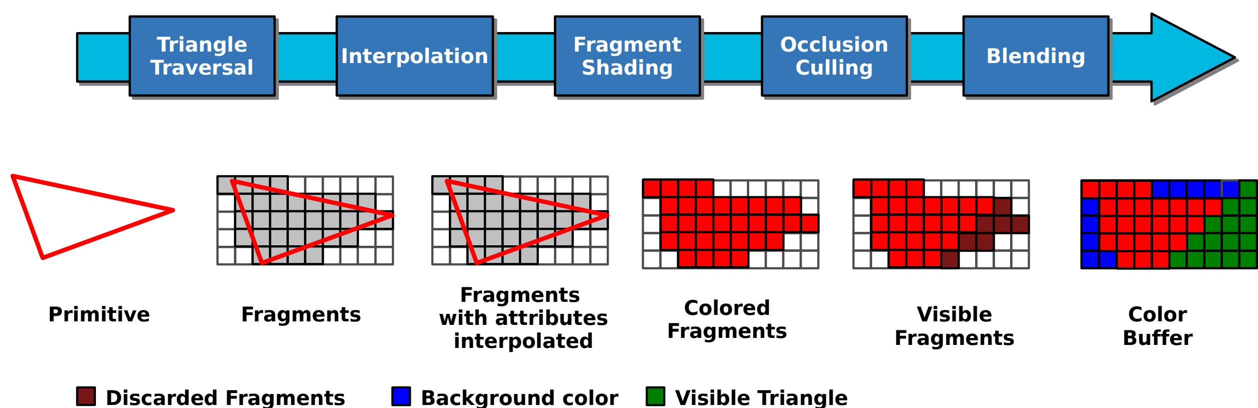


Figure 2.10: Sub-stages of the Rasterization stage of the Graphics Pipeline. In this example, a red triangle is rasterized over a blue background, and some fragments of the red triangle are occluded by a green triangle.

The edge functions have the following properties:

- If $E(P)$ is greater than 0, then P is in the right side of the vector.
- If $E(P)$ is equal to 0, then P is on the line.
- If $E(P)$ is smaller than 0, then P is in the left side of the vector.

Making use of these properties, if the sign of all three equations is positive for a point P, then P is inside the triangle (see Figure 2.11).

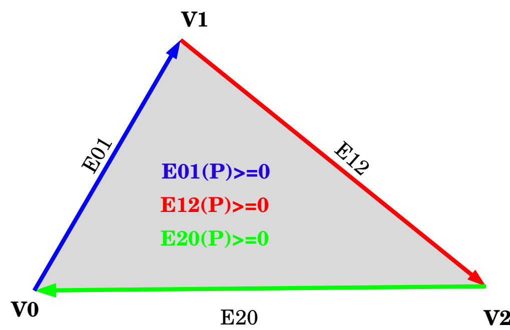


Figure 2.11: Edge functions (E01, E12 and E20) of a primitive defined by vertices v0, v1 and v2.

- **Interpolation:** For every fragment produced for a given primitive, the rasterization stage interpolates the attributes associated to the vertices of the primitive, thus producing a value that corresponds to the fragment. Linear interpolation and perspective corrected interpolation are the most common interpolation modes used for orthogonal projection and perspective projection respectively. For every fragment with position P, three barycentric coordinates -i, j and k- are computed using the edge functions of the primitive:

  1. $i_bary(P) = E01(P)/E01(V2)$
  2. $j_bary(P) = E12(P)/E12(V0)$
  3. $k_bary(P) = E20(P)/E20(V1)$

  Then, the barycentric coordinates are used to interpolate the values of the attributes for the given fragment. For example, when using linear interpolation, the interpolation of the Depth values for a given fragment F inside the primitive defined by vertices $v0$, $v1$ and $v2$ is:

  $Depth(F) = i_bary(F) * V0.z + j_bary(F) * V1.z + k_bary(F) * V2.z$.

- **Fragment Shading:** The rasterization stage computes the color of every fragment produced using the interpolated attributes and other associated data. This process is commonly known as Fragment Shading, and in current systems it is fully programmable and executed in programmable hardware. It is common to access textures in this stage as well as performing lighting.

- **Occlusion Culling:** Commonly known as Visibility Determination, this task consists of determining which fragments are visible and which are not. Although there are other alternatives like the painter's algorithm [26], in most current graphics systems this task is performed by the Z-buffer algorithm [28]. With this algorithm, there exists a buffer called Z-buffer with the same size as the viewport. For every position, the Z-buffer stores the depth (Z-component) of the nearest fragment to the camera. For each new fragment, its visibility is tested against that already stored in the Z-buffer. If the new fragment is visible, the new Z is stored in the buffer, and the fragment passes to the next stage. Otherwise, the fragment is discarded. In Figure 2.12, some fragments of object C are occluded by object A. Likewise, some fragments of object B occlude fragments of object C, while some fragments of B are occluded by A.

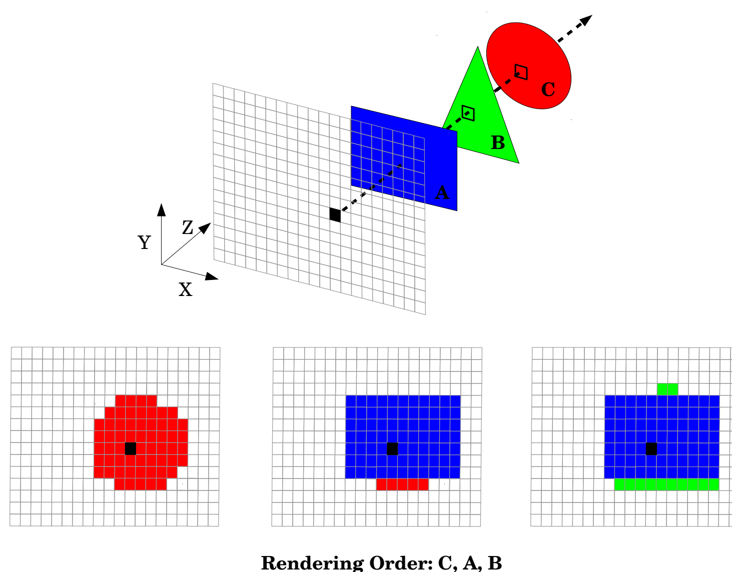


Figure 2.12: Detail of scene where three objects A, B and C are rendered in C, A, B order. The image depicts the fragments of every object that pass the Depth Test.

- **Blending:** The color of the fragments that pass the visibility test is stored in the Color buffer, which has the same size as the viewport. However, rather than simply writing every fragment's color that arrives to this point, they are merged with the color already stored in the Color buffer. Such color has been composed with the color of the fragments that previously passed the test. This color composition is also known as alpha blending and it is meant to allow transparency effects.

For a render step the application stage sends to the geometry stage all the 3D models and other geometry data required to generate a frame. The geometry stage sends primitives to the rasterization stage, which produces an image using the color of the fragments that are visible from the point of view of the camera. Finally, when all the primitives and fragments have been processed, the Color buffer holds a frame that will be read by a display device controller which shows it to a screen.

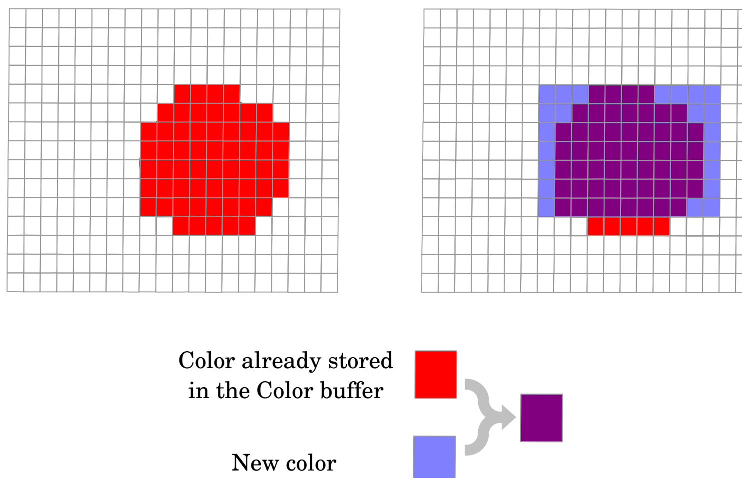


Figure 2.13: Detail of scene where two objects are blended. The image depicts how the fragments of a translucent object are blended with the colors already stored in the Color buffer providing a transparency effect.

This frame can be displayed directly on screen so the input lag is reduced. However, this may cause flickering, tearing and other artifacts which are not acceptable in many scenarios. Generally a double or triple buffering technique is employed to avoid displaying partially updated frames. In the case of double buffering the frame is rendered off-screen in a back buffer, and once the whole frame is rendered, the back buffer is swapped with a front buffer, which is the one displayed on the screen.

2.2 GPU Microarchitecture:

In this section we give a brief overview of the main mobile GPU architectures that we assume as baseline in the experiments we show in following chapters.

2.2.1 Immediate Mode Rendering

Immediate Mode Rendering (IMR) is the preferred mode in desktop, laptop and game console GPUs. In the mobile world, IMR is used, for instance, in NVIDIA Tegra 4 [93]. In IMR, graphics commands are processed in the order they are submitted to the GPU and the corresponding primitives are processed through the entire graphics pipeline stages as soon as they are generated. Due to the fact that the geometry is not guaranteed to appear in front-to-back order, IMR may cause pixel overdraw. This means that the same color-buffer location may be computed and written more than once, which consumes precious off-chip memory bandwidth and wastes energy.

Figure 2.15 shows a block diagram of the IMR GPU pipeline. The Command Processor is the unit in charge of processing the command stream that the GPU receives from the driver. The Command Processor receives the commands from the CPU and it sets the appropriate control signals so the input vertex stream is processed through the graphics pipeline. The Geometry Unit

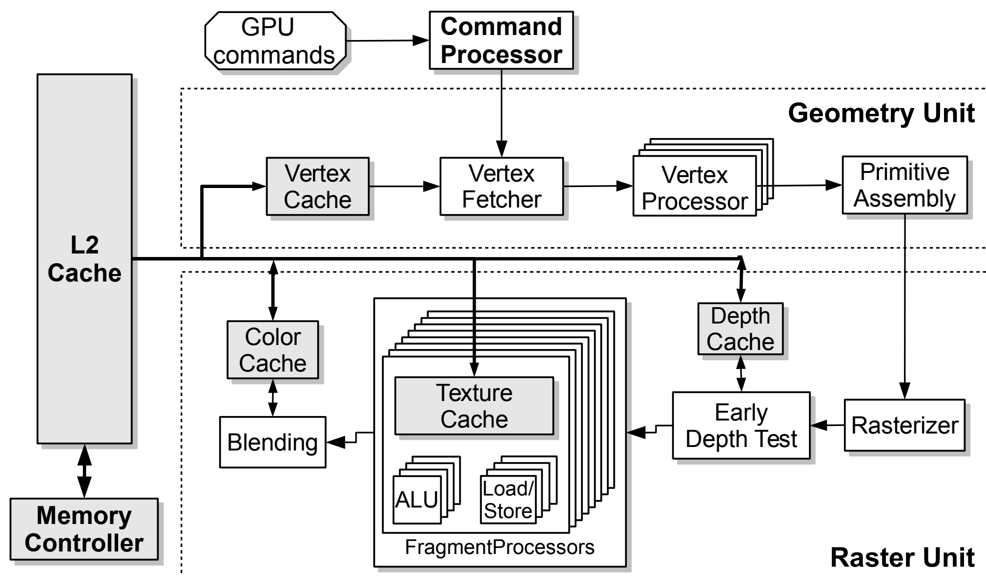


Figure 2.14: Microarchitecture of an IMR GPU.

converts the input vertices into a set of transformed 2D + 1D (depth) primitives in Clip Coordinates space. In the Primitive Assembly stage the stream of vertices is converted to a stream of input primitives (points, lines and triangles). Other operations like clipping, perspective divide, face culling and viewport mapping are also implemented in the Primitive Assembly stage. The assembled primitives that pass Clipping and Face Culling stages are sent to the Rasterizer.

The Rasterizer receives the primitives and discretizes them creating fragments. Every fragment contains interpolated values of every attribute associated to the vertices of the primitive it belongs to. The fragments are tested in the Early-depth test (supported by a main memory Depth Buffer accessed through the Depth Cache). If a fragment is visible (not occluded by a previously tested fragment), it is sent to the following stages of the Raster Unit (otherwise it is discarded). Finally, the fragment colors computed in the Fragment Processing stage are sent to the Blending stage, which combines them with the colors already stored in the Color Buffer.

2.2.2 Tile Based Rendering

Tile Based Rendering (TBR) [106] is the preferred rendering mode in mobile GPUs [89, 38]. In TBR the rendering process is divided into two decoupled pipelines, Geometry and Raster, which are connected through the Tiling Engine. Figure 2.15 shows a block diagram of the GPU pipeline with TBR mode.

The Geometry Pipeline receives vertices and performs geometry-related operations like the ones explained in the previous subsection (transformations, rotations, projections, clipping, culling, etc.), which produce output primitives that are sorted by the Polygon List Builder in tiles [106]. The tiles are stored into the Parameter Buffer, a buffer in system memory that is accessed through the Tile Cache.

Once all the primitives of the frame have been stored, the Tile Scheduler begins to work. For every tile, the Tile Scheduler reads the primitives from the Parameter Buffer in program order and sends them to the Raster Pipeline.

The Rasterizer receives the primitives and processes them creating fragments that are tested in the Early-depth test (supported by an on-chip Depth Buffer). If a fragment is not visible (occluded by a previously tested fragment), it is discarded. Otherwise, it is sent to the following stages of the Raster Pipeline. Then, the Fragment Processors render the fragment's colors, which are composed (blended) with the ones already stored in the on-chip Color Buffer.

Once the tile rendering has been completed, the on-chip Color Buffer is flushed to the Color Buffer in main memory. Note that for TBR pipelines, each pixel color is usually written only once to main memory regardless of object ordering. Only an overflow in the Parameter Buffer causes the GPU to render the already sorted geometry, thus writing main memory more than once, but it is highly uncommon. With TBR, pixel overdraw still occurs but it happens in the local buffer, which saves pixel-related off-chip memory bandwidth with respect to IMR [106]. On the other hand, in TBR the geometry-related memory traffic is increased due to storing and retrieving the geometry of the Parameter Buffer.

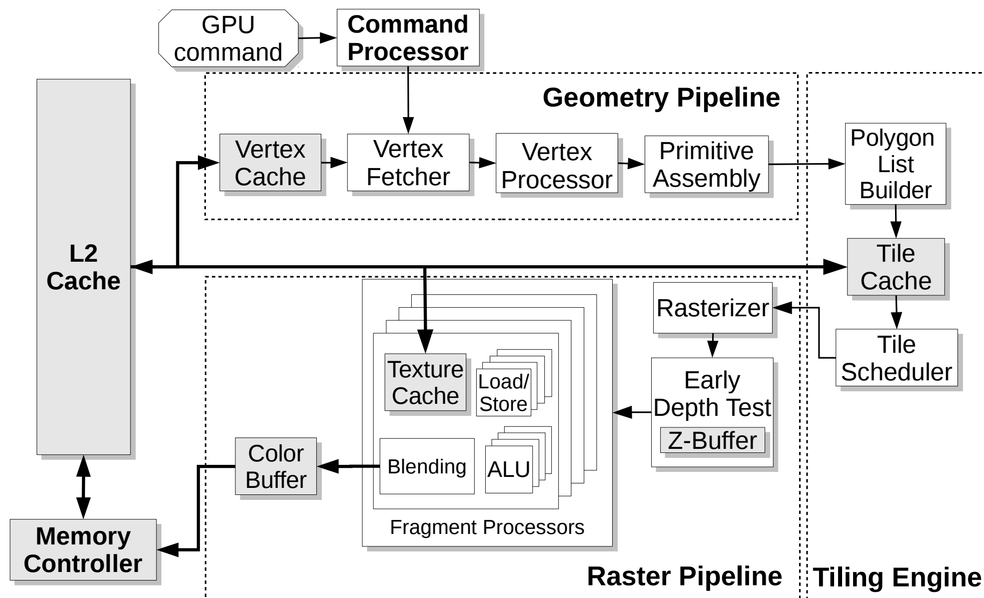


Figure 2.15: Microarchitecture of a TBR GPU.

Deferred Rendering TBR GPU

Deferred Rendering (DR) reduces overshading by first performing the Hidden Surface Removal (HSR) stage, which computes the final state of the Depth Buffer for a given tile. Thereafter, it starts an ordinary rendering of the tile where the Early-depth will discard all the occluded fragments and achieve minimum overdraw. DR not only eliminates overdraw but also guarantees that the Fragment Processor is used only for those fragments that are visible in scene.

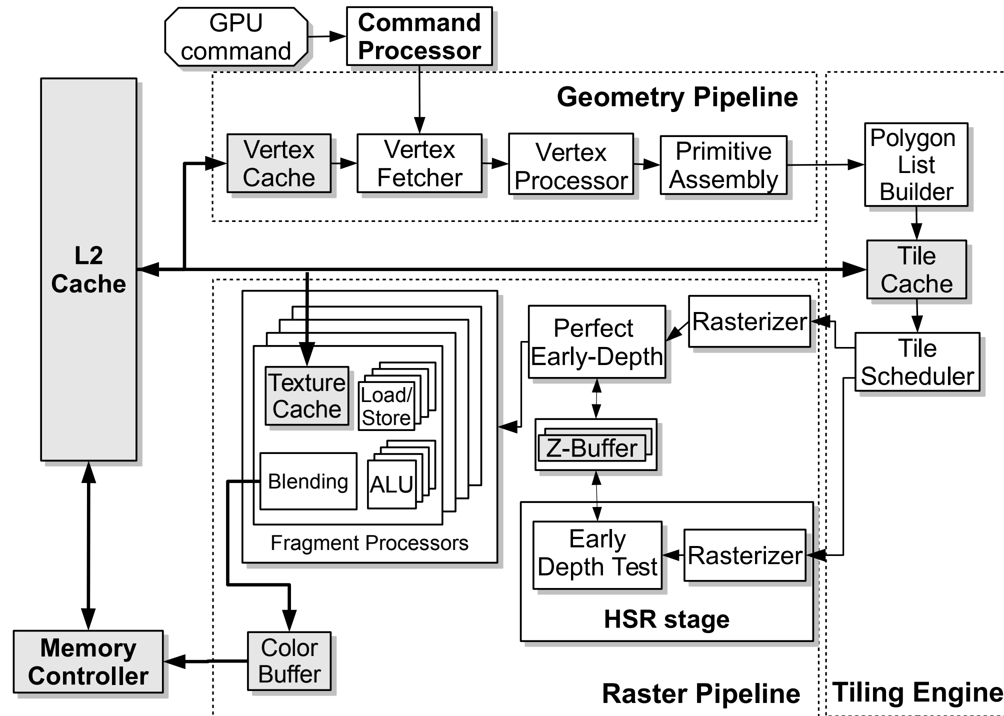


Figure 2.16: Microarchitecture of a TBR GPU implementing Deferred Rendering.

A DR technique has been commercially implemented by Imagination Technologies in their tile-based PowerVR GPU family [91], which they refer to as a TBDR. However, since only partial information about this technique has been disclosed, our Deferred Rendering implementation models what we believe is the most optimistic interpretation of this partial information, in order to be used in the comparisons with our proposal.

A naïve implementation of DR would be to include HSR as a sequential stage in the Raster Pipeline. We implement this naïve implementation and we observe that the execution time increases for every one of the benchmarks tested, 23% on average, while the energy consumption increases around 6% on average when compared with the baseline TBR GPU. In this naïve approach the rest of the Raster Pipeline is stalled while performing the HSR, which badly hurts both performance and energy compared with the baseline GPU. Therefore, it is clear that performing the HSR stage in sequence with the rendering is not an adequate implementation. However, those huge overheads can be removed by performing the HSR stage in parallel with the other stages of the Raster Pipeline (see Figure 4.8). Thus, in this optimized scheme, while the HSR is being executed for tile i+1, the rest of the Raster Pipeline is executed in parallel to render the tile i. Obviously, this parallel implementation introduces a hardware cost and some hardware blocks, such as the Rasterizer, the Early-depth test and the on-chip Depth Buffer, are replicated. Furthermore, the Tile Scheduler is equipped to handle memory requests of two primitives in parallel: one primitive from the tile being rendered and the other one from the tile in the HSR stage. This does not mean that the Tile Cache has now two read ports, but the Tile Scheduler will arbiter between both request queues and only one will be sent to the Tile Cache each cycle in a Round Robin fashion.

3

Methodology

This chapter presents the simulation infrastructure we use in this dissertation for estimating performance and energy consumption of the GPU and the CPU. First, we introduce the simulators we utilize to perform the experiments. Then, we present the workloads we employ in such experiments.

3.1 Simulators

In this section we first give a brief overview of the main existing simulators for GPU and we introduce Teapot [111], the simulation framework that we employ to evaluate the proposals included in this work. Then, we briefly describe Marss86 [173], a multicore simulation environment for the x86-64 ISA that we employ to evaluate performance and energy of several CD algorithms run on a multicore CPU.

3.1.1 GPU Simulation

There are several GPU simulators widely employed in the architecture community that model different aspects of desktop-like GPUs. Some of them, like GPGPUSim [116] and Barra [126] are targeted to simulate General Purpose-GPU (GP-GPU) architectures, so they support OpenCL [192] and CUDA [128]. GPGPUSim is the most widely accepted GPGPU simulator in academia with an overwhelming number of hardware designs evaluated using it. Despite the popularity of GPGPUSim, it is not tailored to run graphics APIs like OpenGL [52]. GPUWattch [162] is an energy model based on McPAT [163] developed for GPGPUSim. Attila [133], QSilver [188] and Multi2Sim [193] provide support for OpenGL, but they do not provide support for OpenGL ES, so they cannot run mobile graphics workloads. Attila includes a generic GPU microarchitecture that is claimed to

closely track the hardware features of GPUs around 2006. Adaptable and configurable, it can be used to evaluate multiple hardware configurations for desktop- and embedded-like GPUs. However, it focuses on IMR GPUs and does not include a Tile-Based Rendering mode architecture, much more popular in embedded devices. GRAAL [155] models a Tile-Based Rendering architecture, includes a power model and provides OpenGL support. Qsilver also provides OpenGL support and includes a power model but it models Immediate-Mode Rendering GPUs. Despite OpenGL support is given in some of the aforementioned simulators, OpenGL ES [57] is not available in any of them so they cannot run the plethora of graphics workloads currently available for embedded devices like smartphones and tablets.

Teapot

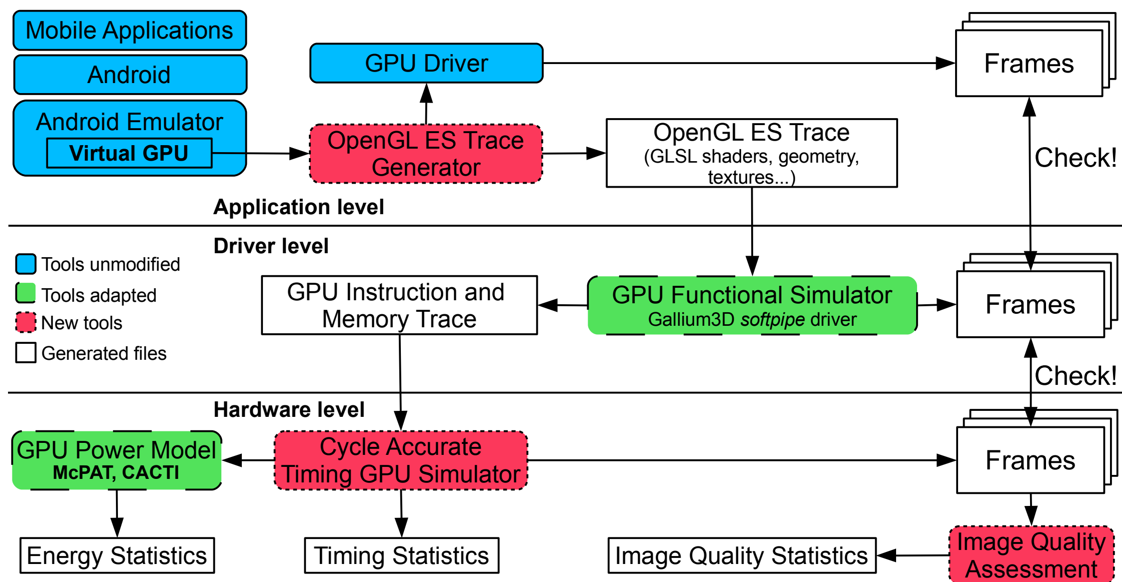


Figure 3.1: Overview of Teapot simulation infrastructure.

Our research group develops Teapot [111], a simulation framework that provides support for OpenGL ES API. Teapot is a mobile GPU simulation infrastructure that supports full-system GPU simulation and that is able to run Android unmodified commercial mobile graphics workloads and evaluate performance, energy consumption and image quality. Furthermore, Teapot is able to profile multiple applications that access the GPU concurrently. In short, Teapot includes an OpenGL commands interceptor, a GPU trace generator and a cycle-accurate GPU simulator for both IMR and TBR GPU architectures, as well as a power model based on McPAT [163]. As shown in Figure 3.1, the simulation stack of Teapot is comprised of different tools that can be categorized in three different architecture levels[1]:

- **Application Level:** Teapot employs the Android Emulator included in the Android SDK [87] for running unmodified commercial applications on a desktop computer. Based on QEMU2 [74],

[1]A more detailed description is available in the dissertation of Arnau [108]

the Android Emulator supports GPU hardware acceleration, which allows to execute state-of-the-art graphics workloads at real-time frame-rates. When enabling GPU hardware acceleration, the OpenGL ES commands issued by the software running in the emulator are not processed inside the emulator but are sent to the host GPU driver.

The OpenGL ES trace generator is a library that interposed between the Android emulator and the host GPU driver is able to intercept all the OpenGL commands sent by the emulator, stores them into an OpenGL ES trace file and finally redirects them to the host GPU driver. The OpenGL ES trace file contains all the necessary data to reproduce the original OpenGL ES command stream (GLSL shaders, textures, geometry and OpenGL state information). To generate an OpenGL ES trace file the user simply runs an application on the Android Emulator and while the application is being executed the trace of OpenGL commands is being generated.

- **Driver Level:** Teapot is a trace driven simulator that employs a GPU functional simulator to generate a trace that includes all necessary information to later perform a GPU timing simulation. The GPU functional simulator of Teapot consists on an instrumented version of *softpipe*, a software renderer included in Gallium3D [88]. Gallium3D is an infrastructure for building 3D graphics drivers that allows portability to all major operating systems.

  The modified software renderer in Teapot is fed with an OpenGL ES trace file generated in the previous level. Every OpenGL ES command is executed as usual, but additionally the instrumented version of *softpipe* gathers information about the rendering process and generates the GPU Instruction and Memory Trace. The generated trace file includes vertices, primitives, fragments, texels, samplers, vertex and pixel shader programs (translated to TGSI assembly language [84]), as well as the memory addresses to read/write geometry, textures and framebuffers (Color and Depth buffers).

- **Hardware Level:** The trace generated in the previous level is given to the cycle-accurate GPU simulator, which gathers activity factors for all the components included in the timing model of the GPU. Teapot provides two baseline GPU architectures: IMR and TBR (see Figure 3.2). The mobile GPU model that is assumed in Teapot closely tracks the ultra-low power GPU in the NVIDIA Tegra chipset [70] for IMR and Mali-400MP [89] for TBR, but they are highly configurable and it can model GPUs with different number of processors, cache sizes and associative schemes, among others.

  - **System memory:** Teapot employs DRAMSim2 [182] to simulate system memory. DRAMSim2 is a cycle accurate simulator that includes SDRAM, DDR SDRAM, DDR2 SDRAM and DDR3 SDRAM memory models including a DRAM memory controller, DRAM ranks and banks, as well as the buses they use for communication purposes.
  - **Power model:** Teapot uses McPAT [163] to compute the static and the dynamic energy consumed by the main hardware structures of the GPU: processors, caches, queues, register files and prefetching tables among others. However, the power model of the functional units is based on the one used in the Qsilver simulator [188]. Furthermore, based on the energy model proposed by Pool et al. [176], we have extended the power model of Teapot by including the rasterizer.

    The GPU timing simulator calls to McPAT, which employs all the micro-architectural parameters (number of processors, type of processor, caches and caches size, etc) and

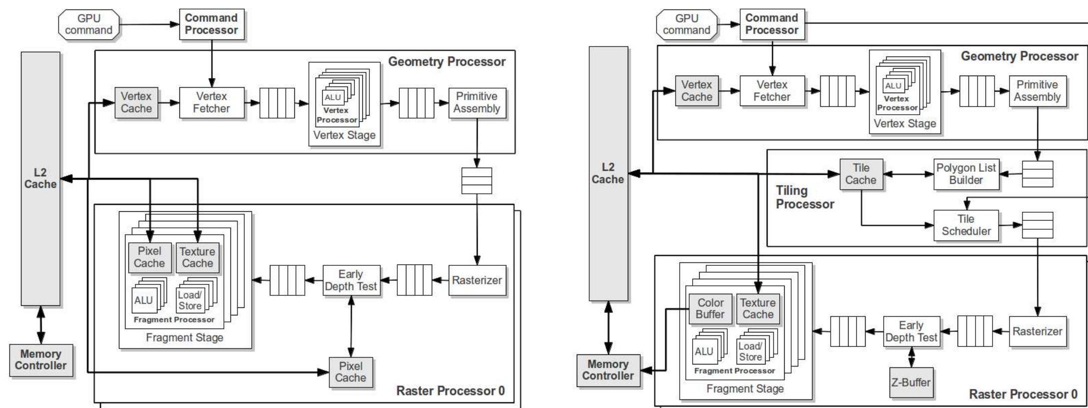


Figure 3.2: NVIDIA Tegra like architecture (left), Mali 400 MP like architecture (right). Images from teapot paper [111].

builds an internal representation of the hardware. McPAT computes the leakage of every hardware structure and the energy required to access such structures. As the GPU timing simulator gathers activity factors of all the components included in the timing model of the GPU, those activity factors of components also included in the power model are sent to McPAT to compute the dynamic energy consumption. The total dynamic energy is obtained by multiplying each activity factor by the energy cost estimated by McPAT for the corresponding hardware structure. The static energy is computed by multiplying the total GPU leakage by the execution time. Teapot produces per frame and global energy consumption and timing reports.

– **Image Quality Assessment:** Teapot generates and stores the frames of the workloads analyzed in three different components of the infrastructure. First, when the OpenGL ES trace generator stores the OpenGL ES commands in a file, the frames generated by the real hardware are also stored. Second, when the functional GPU simulator (*softpipe*) is executed to generate a GPU trace, it also produces the corresponding frames of the trace. Finally, when the cycle-accurate timing simulator executes the GPU trace the corresponding frames are also generated. Among other uses, the Image Quality Assessment module is employed to check correctness of the whole process.

Improvements made to Teapot

During the development of this thesis changes and improvements were made to Teapot in order to carry out the studies presented in this document. The following ones are some of the changes which we consider to be more relevant for the context of this document:

- **OpenGL calls interceptor:** We modified the OpenGL calls interceptor included in Teapot to work with the driver of an NVIDIA GTX 970. This modification allowed us to increment the frames per second obtained with Teapot for our set of benchmarks.
- **Vertex positions:** The GPU trace produced by Teapot did not include the vertex positions

of the input vertices. We modified the Gallium's *softpipe* version of Teapot so it stores them too in the GPU trace. With this addition we are able to obtain the geometry of the scenes included in the GPU trace, which we use in Chapter 5 to study the Collision Detection on a CPU.

- **Face Culling Stage:** The previous model of the Primitive Assembly Stage of Teapot did not explicitly include Face Culling. Let us name as input primitives the primitives created in the first step of Primitive Assembly and as output primitives the primitives that are tested in the Face Culling stage. In the previous model of Teapot, only the "not culled" output primitives were being stored in the GPU trace. Likewise, the cycle accurate GPU simulator was only accounting the cost of Face Culling for "not culled" output primitives. Given that the number of primitives that are culled by Face Culling is significant, sometimes greater than the not culled primitives, we modified both the Gallium's *softpipe* version of Teapot to store all output primitives and the cycle accurate GPU simulator to include a new Face Culling model that account the cost of culling primitives. Furthermore, with these modifications we were able to model the Deferred Face Culling used in Chapter 5.

- **First Level Caches and Buffers:** We included the Vertex Cache, Tile Cache as well as Color Buffer and Z Buffer in the power model of Teapot.

- **New Simulation Models**: We have included new simulation models into Teapot for Tile Based Deferred Rendering, Unified Shader Architecture (for TBR GPUs), as well as a mode to emulate software Z pre-pass.

- **Functional simulator for TBR GPUs**: We added a functional simulator for TBR GPUs that allows rapid design and evaluation of new techniques.

- **TeaTools**: We created an heterogeneous set of tools that work with the GPU traces created by Teapot and allows to visually quantify overshading/overdraw, order among objects, detect collisions among objects and inspect frame creation drawcall by drawcall.

3.1.2 Collision Detection CPU Simulation with Marss86 and Bullet

In the work presented in Chapter 5 we study Collision Detection algorithms on a CPU. In order to estimate the time and energy consumption of such algorithms executed on a CPU we employ Marss86 [173]. Marss86 is a full-system simulation framework to simulate/emulate x86 multicore systems with detailed pipeline model including unmodified operating systems, kernel interrupt handlers and standard libraries (see Figure 3.3). Marss86 is highly configurable and allows to simulate multiple out-of-order/in-order cores and memory models. Marss is based on PTLsim [199] and runs on top of the QEMU [74] emulation environment. We employ Marss to model a processor equivalent to a dual-core ARM Cortex-A9 processor [94] (see Table A.2).

In order to simulate the CD algorithms we employ Bullet [129], a 3D Real-Time Multiphysics Library. Bullet provides state-of-the-art collision detection for soft and rigid body dynamics. Furthermore, Bullet is widely used in industry [95] (e.g., Grand Theft Auto V and Red Dead Redemption). For a given GPU trace we obtain the 3D meshes of vertices of every collisionable object in the same world space coordinates as they are in the original benchmark. With this

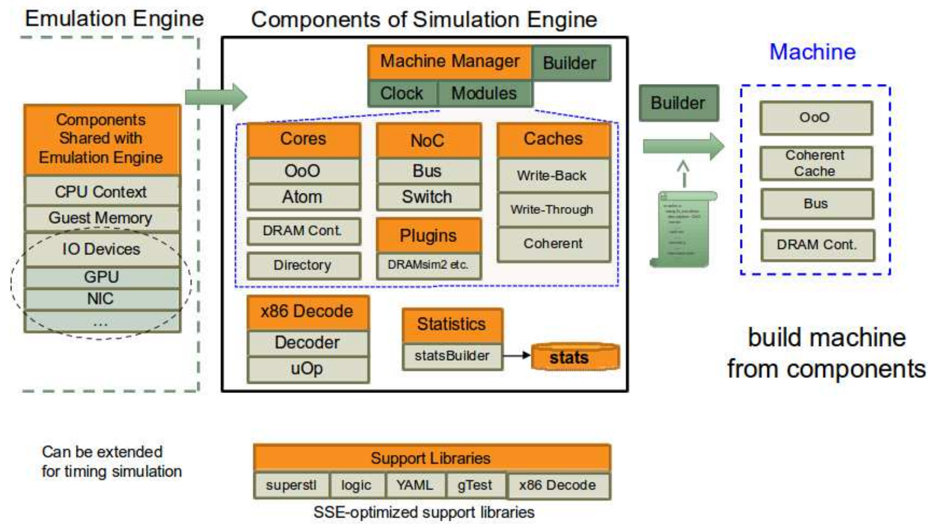


Figure 3.3: Overview of Marss components. Source: www.marss86.org.

information a CD algorithm is able to the test overlaps in a given scene. Then, an application loads the meshes of the collisionable objects in a scene and using Bullet it performs the CD for every frame of the original benchmark included in the GPU trace. We implement two different CD versions, the first one just performs a broad phase, and the second one performs both broad and narrow phases. These two versions are simulated with Marss and the activity factors generated are fed into McPAT to obtain the energy cost of performing the CD in the CPU. The time and the energy of loading the 3D meshes are subtracted from the CPU results.

Table 3.1: CPU Simulation Parameters.

CPU	
Tech Specs	1500 MHz, 1 V, 32 nm
2 Cores	1 MB L2
Core Parameters	
Execution mode	in-order
CPU Architecture	Harvard
L1 Instruction Cache	32 KB/Core
L1 Data Cache	32 KB/Core

3.2 Benchmarks

Our set of benchmarks is composed of nine commercial Android applications with different graphics characteristics. Although there exist splendid commercial benchmarks like GFXBench [96], we prefer to rely on real games because the commercial benchmarks tend to exacerbate the number of advanced 3D effects in order to stress the hardware features of mobile devices regardless the energy consumption of the device. The main target of such benchmarks is to rank the performance

of the hardware devices where they run, so energy consumption is not an issue. On the contrary, mobile games tend to be simpler than those cutting edge benchmarks in order to enlarge battery life. In fact, battery life is one of the most important satisfaction factors for users [32]. Furthermore, a significant number of users uninstall [76, 80, 78, 79] and 36% of users stop using an application [77] if it turns out to be a battery killer application.

The workloads included in our set of benchmarks are representative of the mobile graphics application ecosystem as it includes popular Android games for smartphones and tablets. Furthermore, the applications included in our set of benchmarks employ most of the features available in OpenGL ES1.1/2.0 graphic APIs [101, 100].

Despite Khronos released OpenGL ES 3.0/3.1/3.2 specifications in December 2013, March 2014 and August 2015 respectively [97, 98, 99], during most of the time of this thesis there were not available representative Android games using OpenGL ES 3.XX API. Furthermore, by the time of doing the research studies included in this dissertation Gallium (*softpipe*, *llvmpipe*, *swr*) did not include support for a huge number of features included in OpenGL ES 3.XX. To the best of our knowledge by the time of writing this dissertation the current state of *softpipe* is that it does not support OpenGL ES 3.XX. However, we hope that forthcoming developments on Gallium support for OpenGL ES 3.0/3.1 will allow us to include support for them in Teapot in the future (see future work in Chapter 6.2). In following sections we present and briefly characterize our set of benchmarks.

3.2.1 Benchmarks Set

As graphics animation applications are one of the most popular categories in the main application markets, 3D games are the workloads that exploit the most advanced features of embedded GPUs. We focus on them because they represent a significant and growing market in the mobile segment and the trend is towards more complex and realistic scenes and effects. However, it is true that 2D games also represent an important part of the market. Despite that, we did not include 2D games because the nature of the techniques proposed in this work makes 3D games the applications with greater potential. Take into account that 2D games are usually based on back-to-front rendering and Alpha Test in order to provide nice borders to the objects in the scene (sprites), and one must respect the rendering order in most of the cases. On the other hand, 2D games usually employ uniform grids to perform the CD, where the memory complexity is proportonal to the dimensions of the scene and the computational cost is usually constant. Nevertheless, we include the evaluation of 2D workloads as future work in Section 6.2. Regarding the selection of frames, it is worth to mention that for both techniques we include sequences of frames that represent the most typical use-case in those games. For example, when we study CD the trace includes frames before collisions occur, frames while the collision is produced, and frames after the collision has occurred. Figure 3.3 shows screenshots of our workloads. Below we give a brief description of every game in our benchmark set:

- **300: Seize Your Glory.** Hack & slash game with dynamic camera movements where the player must combat different enemies across several levels while moving around a 3D scene. The game displays high definition 3D graphics and features advanced effects like water rendering, smoke, fog, rain, fire, blood splatters, among others.

(a) 300: Seize Your Glory

(b) Captain America: Sentinel of Liberty

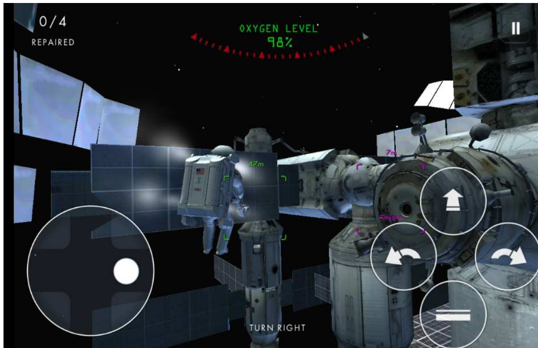
(c) Air Attack

(d) Crazy Snowboard

(e) Forest 2

- **Air attack.** Winner of Unity Awards 2010, Air Attack is an scroll flight combat arcade that features high definition 3D graphics including LightMaps, SpecularMaps and particle systems. Air Attack employs real-time physics and destructible objects like bridges and buildings. The player controls the plane top/bottom/left/right around the scene, which scrolls bottom-top through the entire level.

- **Captain America: Sentinel of Liberty.** Beat'em up action game with dynamic camera movements where the player controls Captain America while he runs, jumps, slides and



(f) **Gravity: Don't Let Go**

(g) **Sleepy Jack**

(h) **Temple Run**

(i) **Shoot War: Professional Striker**

Figure 3.3: This figure shows a screenshot for each of the Android games included in our set of benchmarks.

fights different enemies with his shield. Captain America employs programmable shaders to implement advanced visual effects like explosions and smoke among others.

- **Crazy Snowboard.** Snowboard action game where the player controls a rider that jumps and makes tricks like flips, spins, grabs or grinds over different objects as he descends the track. Crazy snowboard includes bump maps, lighting and basic terrain models.

- **Forest 2.** Haunting horror high definition game where the player controls a character that moves around a forest and whose quest is to escape and banish a white ghost with long black hair. The game includes improved AI and uses programmable shaders that feature basic terrain models and a rich scenario with a great number of objects.

- **Gravity: Don't Let Go.** Action game with dynamic camera movements that features high definition 3D graphics. The player controls an astronaut in a space suit that must accomplish different missions as struggles to survive in the zero-gravity environment of the vast space. The game employs real-time physics and programmable shaders that implement visual effects like the propulsion jets of the astronaut propulsion unit.

- **Shoot War: Professional Striker.** First person shooting game with different scenarios where the player must fight with different firearms multiple enemies that exhibit simple AI. The game employs programmable shaders to implement basic terrain models and visual effects like fire and blood splatters.

- **Sleepy Jack.** Visually appealing action arcade game where the player controls Sleepy Jack while he is dreaming as he flies around a tunnel, collects dream bubbles and interacts with other multiple objects and enemies. This game includes destructible objects and employs high definition 3D graphics and programmable shaders to implement advanced visual effects like explosions, particle systems and lighting.

- **Temple Run.** One of the most popular mobile games, Temple Run is an adventure arcade where the player controls an explorer that has just stolen a cursed idol from a temple and now runs for his life escaping from Evil Demon Monkeys that chase him. The player has to jump, turn, swipe and slide to avoid different obstacles, collect coins and buy items. Like all the games included in our set of benchmarks, Temple Run has been developed with Unity3D and includes water rendering and other visual effect like fire and fog.

3.2.2 Benchmarks Characterization

Our benchmarks set is composed of nine Android commercial 3D applications (see Table 3.2), all of them using Unity3D [21]. Most of them are very popular and count with a high number of downloads. In the case of *Sleepy* it is worth to mention that it is not a free application, which greatly reduces the number of downloads. As can be seen the games have been developed and updated in recent years, being *Captain America* the oldest benchmark (updated in 2011).

We now briefly analyze the key characteristics of our set of benchmarks for the Geometry and the Rasterization stages. Table 3.3 shows information of the Geometry stage. The second column reports the average number of instructions executed per processed vertex in the Vertex Shading (VS) stage, expressed in terms of assembly instructions of the Tungsten Graphics Shader Infrastructure (TGSI) ISA [84]. TGSI is the only intermediate assembly language that is employed in all the drivers of Gallium. TGSI is based on the ARB assembly language (created by the OpenGL Architecture Review Board), which is one of the first low-level shading languages for GPUs with the aim to standardize the control of the hardware graphics pipeline. As can be seen the two benchmarks that employ OpenGL ES 1.1 (*Air Attack* and *Crazy Snowboard*) are among the ones with smaller ratio

Table 3.2: Benchmarks Set.

Benchmark	Description	Downloads (M)	OpenGL ES	Updated
300	hack & slash	10-50	2.0	Feb 2014
Air Attack	flight arcade	10-50	1.1	Jan 2014
Captain America	beat'em up	1-5	2.0	Jul 2011
Crazy Snowboard	snowboard arcade	5-10	1.1	Dec 2015
Forest 2	horror	1-5	2.0	Oct 2016
Gravity	action	1-5	2.0	Dec 2013
Professional Striker	shooter	10-50	2.0	Feb 2017
Sleepy Jack	action	0.5-1	2.0	Jun 2013
Temple Run	adventure arcade	100-500	2.0	Oct 2016

of instructions executed. This is not unexpected as for these games the driver emulates with simple shaders the configurable fixed function options of the OpenGL ES 1.1 VS stage. The third column shows the average number of drawcalls per frame. A drawcall is a draw function call of the API (*glDrawArrays* or *glDrawElements* in OpenGL ES 1.1/2.0) that renders a batch of vertices. The fourth column reports the average number of primitives per drawcall, a parameter that when it is too low causes the application to be CPU bound (the CPU is not able to feed the GPU fast enough and the idle time of the GPU increases). The fifth column shows the average number of primitives per frame and the sixth column shows the average number of output primitives per frame. The input primitives are the assembled at the beginning of the Primitive Assembly stage and are sent to the Clipping stage. The resulting primitives after clipping are referred to as output primitives. The last column shows the percentage of output primitives per frame that are culled by the Face Culling stage. As can be seen this stage removes up to 47% of the geometry before the Rasterization stage. Games like *300*, *Crazy Snowboard*, *Gravity* and *Sleepy Jack*, where the weight of sophisticated 3D objects is high, exhibit the higher face culling rate. On the other hand, *Air Attack* and *Forest 2* exhibit the lower culling rates. Both games contain a large amount of single-faced geometry like the terrain and the trees in the case of *Air Attack* and the leaves of the trees in the case of *Forest 2*.

Table 3.3: Geometry Stage Stats.

Bench	VS insns per Vertex	Drawcalls per Frame	Primitives per Drawcall	Primitives per Frame	Out Prims per Frame	% Culled Out Prims per Frame
300	26.69	192.74	468.50	79445.37	48314.12	42
Air	7.59	21.16	141.16	5905.56	1924.63	23
Cap	15.35	22.35	321.73	11993.35	2007.51	33
Crazy	13.48	20.06	144.09	3486.29	1079.32	42
Forest2	45.23	257.19	170.45	52962.56	30773.78	17
Grav	50.67	40.01	1825.99	56019.44	22743.64	47
Sleepy	19.84	32.50	167.06	8325.70	4963.60	44
Striker	8.59	21.31	481.07	8576.55	6132.88	34
Temple	32.47	25.33	565.64	25177.24	4895.81	31

Table 3.4: Rasterization Stage Stats.

Bench	Fragments per Primitive	Attributes per Fragment	FS insns per Fragment	ALU insns per TEX insns	Texels per Fragment	Depth Complexity
300	103.37	5.39	6.98	18.09	4.36	4.97
Air	324.99	3.50	5.23	3.20	8.98	1.64
Cap	166.69	2.81	5.60	5.24	5.89	1.97
Crazy	594.81	4.69	6.29	10.55	4.00	1.23
Forest2	231.88	8.11	17.63	15.21	13.57	4.85
Gravity	28.06	5.20	20.34	23.19	11.41	1.98
Sleepy	78.84	2.22	6.76	20.28	3.23	1.40
Striker	386.19	4.24	7.42	6.22	7.47	2.24
Temple	97.13	4.31	10.78	17.16	10.67	2.05

Once the geometry stage has finished the resulting primitives that have not been culled are processed in the Rasterization stage. The primitives are first discretized into fragments that contain interpolated values of the per-vertex attributes that the primitive includes. Then, the color of the fragments is computed by the fragment shader. Table 3.4 shows information of our benchmarks in the Rasterization stage. The second column presents the average number of fragments per primitive. The third column contains the average number of attributes per fragment. This is a measure of the amount of work to be done by the Rasterizer for the interpolation of the attributes of the vertices of a triangle. The fourth column contains the average number of fragment shader instructions per fragment produced by the Rasterizer. The fifth column shows the average number of ALU instructions per texture fetching instruction. Take into account that in OpenGL ES 2.0 the texture instructions are the only ones that allow the programmer to access memory within the fragment shader, so the higher number of ALU instructions per texture fetching instruction the less memory intensive the workload. The sixth column presents the average number of fetched texels (texture pixels) per fragment. The number of texels fetched per fragment varies depending on the number of textures employed in the shader and the texture filter selected. For one texture, the filters *nearest-neighbor*, *bilinear* and *trilinear* fetch 1, 4 and 8 texels respectively. Anisotropic filtering with 2x, 4x, 8x and 16x filter modes fetch 16, 32, 64 and 128 texels respectively. The last column shows the depth complexity of the scene, i.e., the average number of fragments per pixel executed in the fragment shaders. This parameter provides a measure of the activity performed in the Fragment Shading stage and it is also commonly referred to as overshading.

As can be seen in the second column of table 3.4, on average the number of fragments is two orders of magnitude greater than the number of primitives, which indicates why typically the fragment shading is known to be most consuming stage of the graphics pipeline. This parameter widely ranges from around 28 up to 594 for *Gravity* and *Crazy Snowboard* respectively. It is interesting to note the special case of *Gravity*, which counts with a detailed geometry and an important part of the screen literally empty, i.e., no fragment has been produced to cover those pixels. The third column indicates a significant variation in the number of attributes per fragment that the Rasterizer must interpolate, which range from 2.22 up to 8.11 for *Sleepy Jack* and *Forest 2* respectively. With respect to the number of fetched texels per fragment we also observe a wide

range from 3.23 (*Sleepy Jack*) to 13.57 (*Forest 2*). Likewise, the fragment shaders employed by our set of benchmarks are significantly different, being *Sleepy Jack* and *Gravity* the workloads with less and more intensive fragment shading with 5.23 and 20.34 assembly instructions respectively. It is interesting to note the smaller ratio of shader instructions in the Rasterization stage with respect to the Geometry stage. The number of ALU instructions per texture fetching instruction is also different for the different workloads of our set of benchmarks and it ranges from 3.20 to 23.19 for *Air Attack* and *Gravity* respectively. Furthermore, our workloads employ different texture filters. Games like *Air Attack*, *Forest 2*, *Gravity* and *Temple Run* employ several textures in the shader and trilinear filtering, fetching around 9 texels in the case of *Air Attack* and more than 10 texels per fragment on average in the other cases. Finally, the last column of Table 3.4 shows the overshading of our set of benchmarks, which ranges from 1.23 (*Crazy Snowboard*) to 4.97 (*300*). As we previously said, computing the color of a fragment and writing it in a given pixel of the Color buffer more times than necessary wastes a considerable amount of main memory bandwidth, time and energy [172]. Thus, reducing this unnecessary activity significantly increases the energy-efficiency of the GPU.

4

Visibility Rendering Order: Improving Energy Efficiency on Mobile GPUs through Frame Coherence

Identifying visible surfaces is a requirement in the graphics pipeline for correct image rendering. The most widespread method to resolve visibility at pixel granularity is the Depth Test, which is typically placed at the end of the pipeline. Figure 4.1 introduces a simplified conventional graphics pipeline. The GPU receives vertices and processes them in the Geometry Pipeline, which generates triangles. These are then discretized by the Rasterizer, which generates fragments that correspond to pixel screen positions. Then, fragments are sent to the Fragment Processing stage, which performs the required texturing, lighting and other computations to determine their final color. Finally, the Depth test compares each fragment's depth against that already stored in the Depth Buffer to determine if the fragment is in front of all previous fragments at the same pixel position. If so, the Depth Buffer is conveniently updated with the new depth, and the color of the fragment is sent to the blending stage, which will accordingly update the Color Buffer (the buffer where the image is stored). Otherwise the fragment is discarded.

One big advantage of the Depth test is that it ensures correct scene rendering regardless of the order the opaque geometry is submitted by the CPU. The main drawback is that the color of a given pixel may be written to the frame buffer in system memory more times than necessary (a problem known as overdraw), which wastes a considerable amount of bandwidth and energy [172]. Moreover, when the GPU realizes that an object or part of it is not going to be visible, all activity required to compute its color has already been performed, with the consequent waste of time and energy (a problem known as overshading), especially in the Fragment Processor, which is the most power consuming component of the graphics pipeline [176]. Reducing the overshading produced by non-visible fragments can significantly increase the energy-efficiency of the GPU.

Figure 4.2 shows the overshading for several applications (details on the evaluation framework

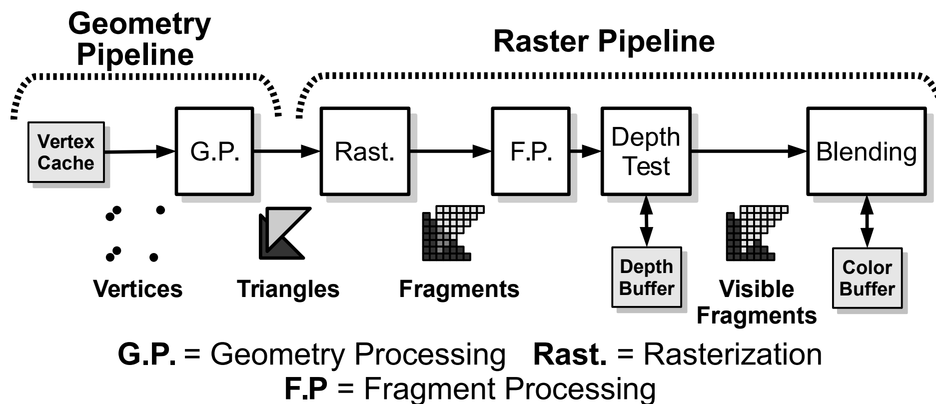


Figure 4.1: Simplified version of the Graphics Pipeline.

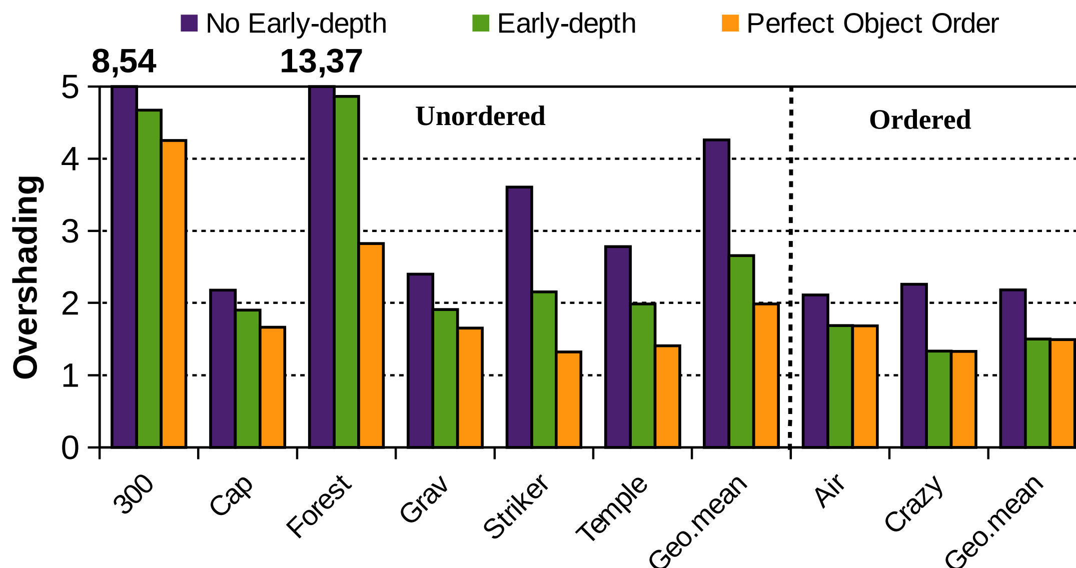


Figure 4.2: Shaded fragments per pixel in a GPU without Early-depth test, with Early-depth test and with perfect front-to-back rendering order at object granularity.

are provided later). We define overshading as the average number of fragments processed per pixel. First bar shows that overshading is extremely high in some applications with complex 3D geometry such as *300* and *Forest 2*, for which each pixel is computed and written around 8.5 and 13 times on average.

Commercial GPU pipelines include an Early-depth test stage that checks fragment visibility before the Fragment Processing, and achieves substantial overshading reductions (see mid bar of Figure 4.2). However, the effectiveness of the Early-depth relies on the software ability to send opaque primitives in front-to-back order, which is not what the software does in most of the cases. The third bar of Figure 4.2 shows the overshading with a perfect front-to-back rendering order at object level granularity, which reduces inter-object overshading to the minimum. As can be seen,

there is significant headroom for improvement, and this is the target of this chapter.

It is well-known that improving the battery life of handheld and portable devices is a major concern for hardware and software developers. Among all the components in smartphone SoCs, the Graphic Processing Unit (GPU) has been identified as one of the top energy consumers [122]. In particular, for graphics applications the GPU has been identified as the principal energy consumer [174]. Further experimental data with the same SoC shows a peak consumption of the GPU 50% higher than the peak consumption of the CPU [30]. The current trend towards more realistic graphics and therefore, more power hungry applications [150] is just aggravating this issue, so, improving the energy efficiency of mobile GPUs is key for future designs [103, 109, 123, 148, 168, 169, 180, 131, 120]. The development of energy-efficient solutions is a requirement to make possible a richer user experience in these platforms.

In this work, we propose a novel hardware technique for mobile GPUs, Visibility Rendering Order (VRO), which tries to render objects in a front-to-back order to maximize the culling effectiveness of the Early-depth test and minimize overshading, hence reducing execution time and energy consumption. Our approach is based on the observation that consecutive frames do not differ much in order to provide the feeling of smooth transition in animated applications. This suggests that the relative order among the objects in frame N is usually the same as in frame N+1 (or very close). Since depth-order relationships between objects are already checked by the Depth Test, VRO incurs minimal energy overheads because it just requires adding a small hardware to capture that information and use it later to guide the rendering of the following frame. This extra activity is performed in parallel with other stages of the pipeline, so no performance overheads are incurred.

For the analysis in this work, we have classified our set of benchmarks into two different groups according to the following. If the reduction in overshading between an ideal front-to-back rendering order at object granularity (third bar of Figure 4.2) and the execution using Early-depth (second bar of Figure 4.2) is smaller than 0.5%, then the benchmark is categorized as "ordered", otherwise the benchmark is categorized as "unordered". Our technique achieves impressive results for the "unordered" group of applications, i.e. those that do not submit objects in front-to-back order to the GPU. For this group, VRO obtains 27% speed-up and 14.8% energy reduction on average when compared with a state-of-the-art mobile GPU presented. For the "ordered" group of benchmarks, VRO achieves minor reductions in overshading, but it neither produces any performance penalty nor energy overhead. Take into account that for ordered benchmarks, VRO may provide benefits to the CPU, because the application does not longer require to order objects front-to-back, which is typically expensive in complex scenes.

4.1 Visibility Determination and Overshading

Current graphics processors implement the Depth Buffer technique to resolve the visibility of opaque surfaces at fragment granularity. During the rendering process, the objects are discretized into fragments that correspond to pixel positions. By testing each fragment's depth against that already stored in the Depth Buffer, the hardware determines if the fragment is in front of all previous fragments at the same position. If so, the Color Buffer and the Depth Buffer are conveniently updated. Otherwise the fragment is discarded. This is commonly known as Depth Test or Z-Test.

Note that the depth and the color of a given pixel may be written multiple times, also known as overdraw, which wastes a considerable amount of main memory bandwidth and energy. This is the case of the Immediate-Mode Renderers (IMR) which is the preferred rendering mode in desktop, laptop and game console GPUs. In the mobile world, IMR is used, for instance, in NVIDIA Tegra 4 [93] and Tegra X1 [71]. With IMR, the graphics commands are fully processed in the order they are submitted to the GPU and the corresponding primitives are processed through the graphics pipeline stages as soon as they are generated, which may cause overdraw if the geometry is not rendered in a perfect front-to-back order. Otherwise, occluded portions of the scene are discarded by the Depth Test and so, the writes to Depth Buffer and the accesses to Color Buffer are reduced, saving precious memory bandwidth. This has motivated some research to sort image objects in the CPU before they are sent to the rendering pipeline. Nevertheless, this approach has important computational costs that make it less appealing for real-time rendering, especially on low power devices.

To further reduce power and latency, most pipelines perform an Early-depth test to fragments before they are sent to the Fragment Processors. Discarding occluded fragments at this pipeline stage saves useless shading and blending work and brings important performance and power benefits. Note that a given pixel may still be shaded multiple times unless opaque surfaces are rendered in front-to-back order, a problem known as overshading. Overdraw and overshading refer to similar problems. Overshading refers to unnecessary executions of the Fragment Processor whereas overdraw represents the unnecessary writes to the Color Buffer, and therefore, whenever there is overdraw there is also overshading. Note that not all overshading is avoidable. Non-opaque or transparent objects will compute the color of their pixels using the color values of the scene behind them, so overdraw for those objects is unavoidable in these cases.

Overshading has also been addressed in other ways. Z-prepass [145] addresses overshading by performing two separate rendering passes with the GPU. First it renders the geometry without outputting to the Color Buffer, just using a null fragment shader, to setup the Depth buffer final values. On a second pass with the real shaders the Early-depth test will perform optimal culling, so overshading will be minimum (just one opaque fragment per pixel will be shaded and written to the Color Buffer). Unfortunately, this approach doubles the amount of vertex processing, rasterization and depth-test work required, which more than offset its benefits. It is only effective for workloads with enough depth and/or fragment complexity where these overheads are compensated by large fragment computation savings, which is not usually the case on mobile applications.

Deferred Rendering (DR) is a hardware technique that avoids overshading through computing the Depth Buffer before starting fragment shading. Currently, DR has only been implemented on Tile Based Rendering (TBR) GPUs [91]. TBR pipelines divide the screen space into tiles and, before rasterization, they assign the geometry of the scene to the tiles, which are then independently rendered. This allows the GPU to use small on-chip memories to contain the Depth and the Color buffers for the entire tile, which dramatically reduces the accesses to main memory [106]. DR adds a hidden surface removal (HSR) phase to the pipeline just before the Early-Depth test. During the HSR phase, all the tile primitives are first rasterized only for position and depth, and the resultant fragments are Early-depth tested to setup the Depth Buffer. Once HSR is complete, the second pass processes the tile primitives as usual along the raster pipeline (they are read, rasterized and depth-tested again), except that this time the Early-depth test performs optimal occlusion culling.

Although the exact details of this technique in commercial systems are not fully disclosed, we have modeled in our framework an efficient implementation of it at the microarchitecture level, which is described in Subsection 4.3.1. In contrast to Z-prepass, DR does not perform the geometry processing twice. However, as can be seen in Figure 4.3, DR still has a non negligible cost: either it introduces a barrier in the graphics pipeline, because the Fragment Processing stage cannot start until HSR has completely finished the tile (see (a) sequential DR), or significant extra hardware is required to perform HSR of tile *i+1* and rendering of tile *i* in parallel (see (b) parallel DR). Further details are given in Section 4.3.1.

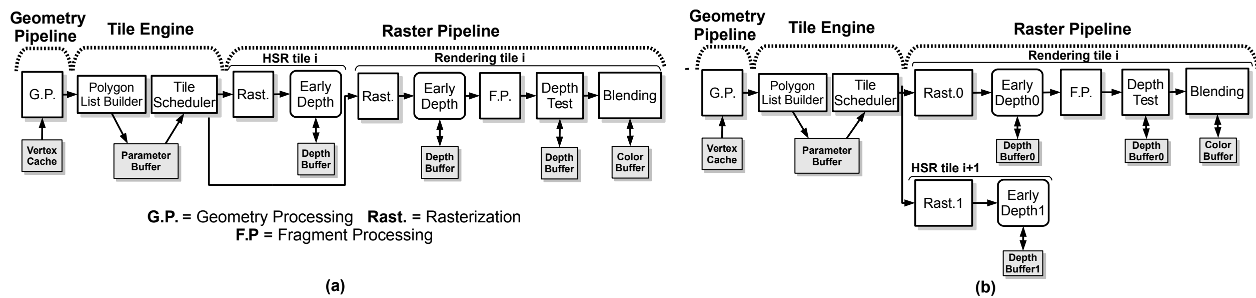


Figure 4.3: Graphics pipeline: (a) Sequential DR. (b) Parallel DR.

4.2 Visibility Rendering Order

To help maximize Early-depth test effectiveness, we propose to record the visibility order of the objects in a frame, assume the same order for the next frame, and then use it to influence the rendering order of the objects in the next frame. This is expected to work since images of consecutive frames normally show a significant degree of similarity to result in a smooth transition among frames, so the ordering of objects in consecutive frames tends to be the same. To produce a quantitative evidence of this, we have evaluated sequences of 50 frames of our benchmarks, and we have observed that the relative order of the objects in a frame closely matches the relative order in the previous frame. Note that in the case that some object is in a different order, a small overshading may occur, but correctness is ensured in any case.

Unlike other approaches, our technique works for all kind of scenes, either static or dynamic, it does not cause CPU-GPU synchronization issues and it has no performance cost because it works in parallel to other stages of the pipeline. This section outlines our technique, and the next section will provide hardware implementation details.

4.2.1 Overview

Figure 4.4 shows the changes to the graphics pipeline introduced by VRO, which will be explained below: the Edge Inserter, the Visibility Sort Unit and the Graph Buffer. Basically, VRO has two stages that operate on consecutive frames:

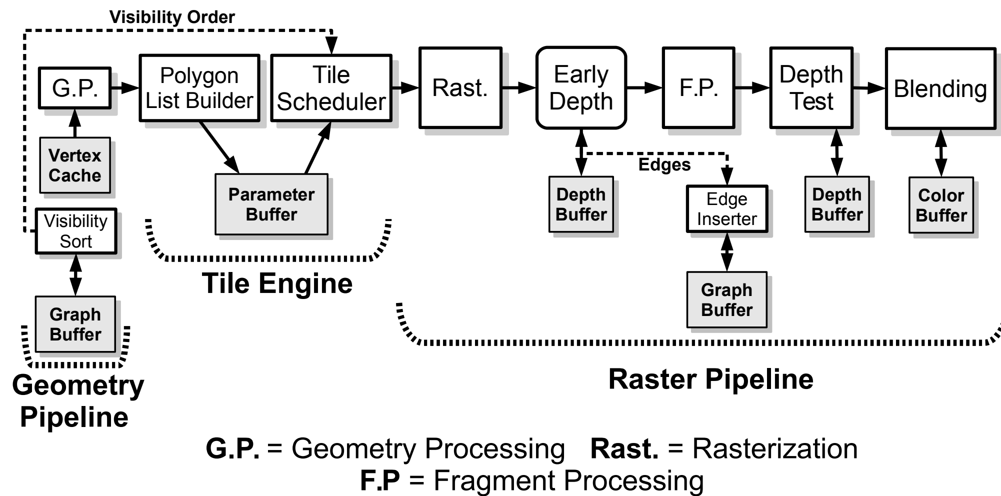


Figure 4.4: Graphics pipeline including VRO.

1. **Creation of a Visibility Graph:** During the rendering of frame *N* the Early-depth test reveals depth precedence relationships between pairs of fragments covering the same pixel position, hence among the corresponding objects. These relationships (edges) are used by the Edge Inserter unit to build a directed graph where objects are represented by nodes, and the edges indicate which objects are in front of others. We will refer to this graph as the Visibility Graph, which is stored in the Graph Buffer.

2. **Creation of a rendering order:** At the beginning of the rendering of frame *N+1*, in parallel with the execution of the Geometry Pipeline, the Visibility Sort unit sorts the Visibility Graph created during frame *N* to generate a depth-ordered list of nodes. We will refer to this list as the Visibility Rendering Order, and it is used by the Tile Scheduler to guide the rendering of the frame *N+1*.

4.2.2 Graph Generation

Figure 4.5 shows an example of the Visibility Graph that is generated for the given frame of Gravity. Let us consider three different objects: the astronaut (object A), the jetpack (object B) and the big solar panel (object C). Let us assume that these objects were submitted and rendered in the order A, B, C. When the object A is rendered ①, the Depth Test annotates in the corresponding areas of the Depth Buffer the depth of A's fragments as well as their identifier. Next, the object B is rendered ② and we know by the results of the Depth Test that, in some pixels, B is in front of A (edge (B, A)). As we explain later, the Depth Test sends this information to the Edges Filter, which filters lots of redundant edges. Finally, the object C is rendered ③ and the Depth Test reveals that A occludes C in some pixels, and B also occludes C in some other pixels, so edges (A, C) and (B, C) are sent to the Edge Inserter through the Edges Filter.

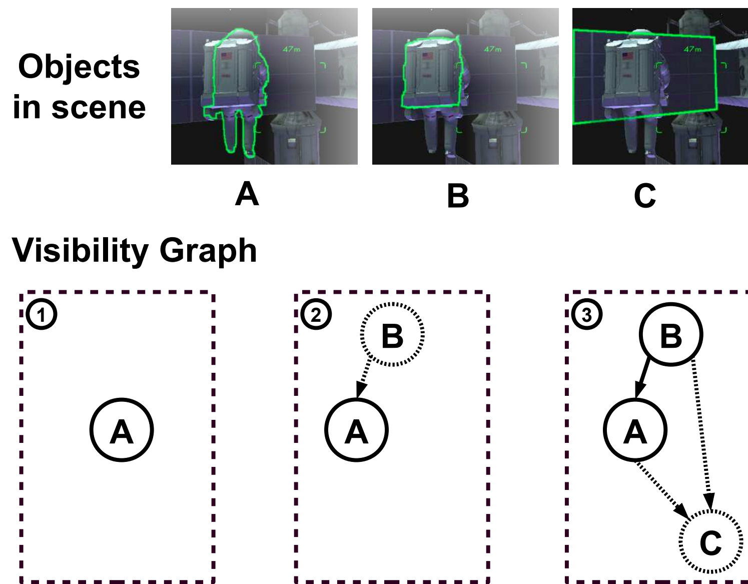


Figure 4.5: Visibility Graph generation for the given scene.

4.2.3 Sort Algorithm

Once the graph is generated, it is sorted to create the Visibility Rendering Order (a front-to-back ordered list of object-ids). Our approach is based on the well known Topological Sort algorithm, first proposed by Kahn [156], which guarantees for DAG graphs (acyclic) that an ordered list of nodes exists and it is generated in linear time. Algorithm 1 outlines the basic algorithm, assuming for convenience that every node is tagged with its number of incoming edges (the in-degree). Nodes with no incoming edges are referred to as roots.

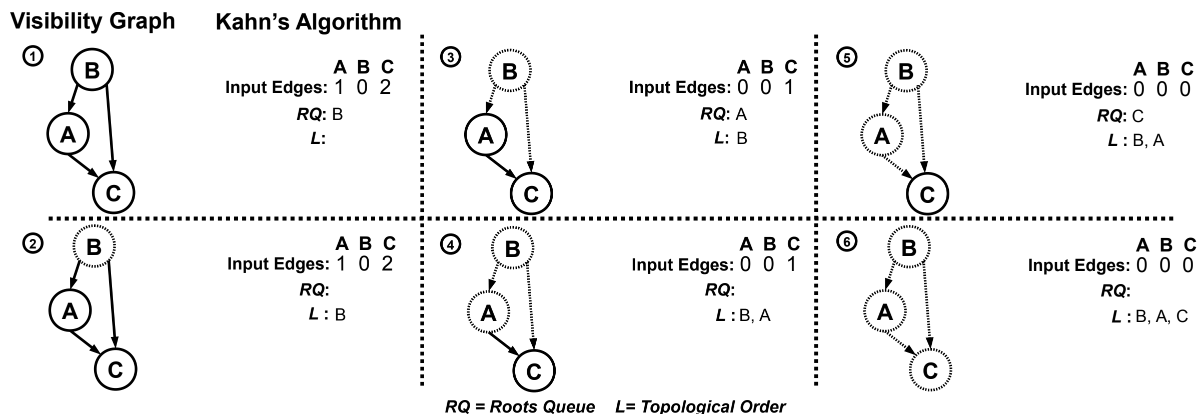


Figure 4.6: Sorting a Visibility Graph with the Kahn's algorithm.

Let us illustrate how a Visibility Graph is sorted with the Kahn's algorithm (see Figure 4.6). The Kahn's algorithm maintains an array with the number of input edges of every node in the graph. As we said, the roots are pushed into RQ and the topological order is stored into L. Initially the graph contains one root, the object B ①. Once the object B has been sorted ②, the input edges

Algorithm 1 Kahn's algorithm.

```
 1: function KAHN(L,RQ)                    ▷ L: empty list that will contain the sorted nodes
 2:                                        ▷ RQ: queue with all initial root nodes of Graph
 3:     while RQ is non-empty do
 4:         remove head node n from RQ
 5:         insert node n into list L
 6:         for child m of n do
 7:             remove the edge from n to m
 8:             decrease the in-degree of m
 9:             if m is a root then
10:                 add m to tail of RQ
11:             end if
12:         end for
13:         remove node n from Graph
14:     end while
15:     if Graph has nodes then
16:         return error                   ▷ Graph has at least one cycle
17:     else
18:         return L                       ▷ a topologically sorted order
19:     end if
20: end function
```

of its children nodes are updated ③, which turns the object A to be a root. Therefore, it can be sorted ④ and the input edges of its child C are conveniently updated. As soon as the object C becomes a root ⑤, it can be sorted ⑥ producing a topological order for the given directed graph.

However, if the graph contains a cycle, this algorithm finishes with an error condition because at some point none of the graph nodes that remain to be sorted have zero in-degree. We found that these cycles are quite common, and actually none of our benchmarks creates a DAG. To cope with this situation, we distinguish three kinds of cycles and apply different solutions in each case:

1. **Auto occlusions between parts of the same object (auto-cycles)**. They are removed by discarding their corresponding edges in the process of creation of the graph in the phase one of the sort algorithm. These edges can be ignored because VRO reorders at object level and the auto-occlusions just contain intra-object precedence relations.

2. **Pairs of interlaced objects occluding each other (parallel-cycles)**. VRO eliminates the cycles created by pairs of interlaced objects by adding to the Visibility Graph only the first precedence relation between two objects (A, B), i.e. A occludes B, found by the Depth Test. If later on a (B, A) relation is found in the Depth Test, it is just ignored and not added to the graph. This is also done in the phase one of the algorithm.

3. **Three or more objects alternately occluding one another (indirect-cycles)**. Since these cycles may be extremely costly to detect, we adopt a cost-effective approach which does not attempt to eliminate them from the graph. We rather extend the Kahn's algorithm to side-step a cycle-induced wrong termination: whenever the RQ is empty and there are still

nodes to be sorted but none of them is a root, we select from the remaining graph nodes the first node in program rendering order among those nodes with minimum number of input edges. Then the node is removed from the graph and added to the tail of RQ. This process is done in the phase two of the sort algorithm. In our experiments, the heuristic that selects a node to avoid cycle-induced wrong termination of the Kahn's algorithm is executed around 13% of the times.

4.2.4 Heuristics to Sort the Objects in a Scene

In this section we present different heuristics to handle the cycles that appear in the Visibility Graph of a given scene. One solution consists on simply create a directed and acyclic Visibility Graph and sort it using the regular Kahn's algorithm. Other solution is to create a Visibility Graph with some cycles and break them appropriately when sorting the graph. We test different heuristics that generate different approximations to the optimum rendering order for the given scene.

Eliminating all the Cycles before Sort Time

Eliminating all the cycles of the Visibility Graph before sorting it allows to employ the regular Kahn's algorithm to obtain a Visibility Rendering Order. The cycles are eliminated by removing a set of edges from the directed graph. Such set of edges is often referred as feedback arc set. Furthermore, a feedback arc set with minimum weight is known as the minimum feedback arc set, whose computation is a problem known to be NP-hard for different kind of graphs [119, 175].

We implement a greedy algorithm that computes a feedback arc set that is an approximation to the minimum feedback arc set, which allows us to eliminate all the cycles of the Visibility Graph before sorting it through the Kahn's algorithm. In this algorithm every edge (A, A) is discarded in the Depth Test. Every edge (A, B) is annotated with the number of fragments that a given object A occludes an object B. The algorithm computes the set E of annotated edges that form the graph G(E), eliminates the cycles creating a DAG, and finally returns a Topological Order of the DAG. This is the algorithm employed to obtain the perfect-object-order shown in the introduction of this chapter. It works in two main phases:

1. The set E of annotated edges that form the graph G(E) is computed in the Depth Test.
2. The algorithm eliminates the edges of the set E that form cycles in the graph G(E):
   (a) The greedy algorithm sorts the set of edges E in descending order of the number of occlusions and creates a list L. A new empty set of edges E' is created.
   (b) For every edge e of L (in order):
      i. If the graph formed by the edges of E' plus the edge e remains acyclic, insert the edge e in the set E'.
      ii. Otherwise, the edge e is discarded.

(c) Once all the edges of the ordered list L have been processed, the resulting graph G'(E') is a DAG. This DAG is fed to the Kahn's algorithm which produces a Topological Order, that is, an ordered list of the objects of the scene.

Eliminating some Cycles at Sort Time

We have tested different schemes that create a Visibility Rendering Order. In all the schemes tested, auto-cycles are eliminated by discarding the edges with same objects as source and target in the Depth Test. Parallel-cycles are handled in two different ways:

1. One can attach a counter to the edges and count the number of times A occludes B, and vice versa. Once the scene has been rendered, we are able to break the cycle by eliminating the edge of each parallel cycle with minimum value, that is, we eliminate the edge that would lead to a higher overshading. This scheme is referred to as C, where C means "counter". We tested two sizes of counters: 32 and 16 bits.
2. On the other hand, one can just add to the Visibility Graph the first edge that arrives to the Edge Inserter. The schemes implementing this option are referred to as PO, where PO means "program order". Obviously, this approach is simpler than the former.

Regarding the rest of the cycles in the graph, they are not eliminated but handled appropriately by an heuristic which selects the next node to visit when none of the remaining nodes is a root (i.e. has no incoming edges). Two different heuristics have been studied:

1. **H1**: This heuristic selects the next node in program rendering order among the nodes with minimum in-degree. We observe that H1 results in less overshading in general. Of course, the effectiveness of this approach depends not only on the in-degree but also on the occluded area which is the real source of the overshading.
2. **H2**: This heuristic selects the node with minimum average depth. We test different versions of H2, selecting the node with minimum maximum-depth and minimum minimum-depth. Both alternatives produce almost identical results.

We also test H1 and H2 alone without the "counter" or the "program order" mechanisms, so the Visibility Graph is traversed as it is produced in the Depth Test. The different versions of VRO are summarized in Table 4.1, and evaluated in subsection 4.5.2. Note that the combination PO_H1 is the one described in subsection 4.2.3 and it is assumed for all our experiments unless stated otherwise.

Our main goal is to improve performance while still reducing energy consumption on a mobile GPU. VRO achieves both goals by improving the effectiveness of the Early-depth to cull hidden surfaces before they reach the Fragment Processing stage, so that the number of shader instructions executed is reduced. We have tested several heuristics to handle the cycles of the Visibility Graph that approximate this goal with different performance/energy trade-offs. The suitability of one or

Table 4.1: VRO alternatives.

Version	Par. Cycles	Heuristic
C_H1	counter	min in-degree
C_H2	counter	min average depth
PO_H1	program order	min in-degree
PO_H2	program order	min average depth
MinZ	-	min average depth

another heuristic on a given hardware platform will greatly depend on design issues. Bear in mind that, regardless of the approximation image correctness is guaranteed in any case by the depth test. Our choice here is a heuristic that implies a cost effective implementation of VRO and clearly illustrates its feasibility and effectiveness.

4.2.5 Partial Order of Objects

The Depth Buffer only stores the depth of one fragment at every pixel position. Thus, when a new fragment is tested the comparison is performed between the new fragment and the one visible so far, so there may be objects whose fragments are never compared. Therefore, this comparisons will provide just a partial order of the objects. Hence, one may wonder whether the missing node relationships may lead to build a wrong Visibility Graph. To answer this question, note that the relative render ordering of two objects is only relevant for visibility purposes if they overlap at some region, and that region is visible at least in one pixel.

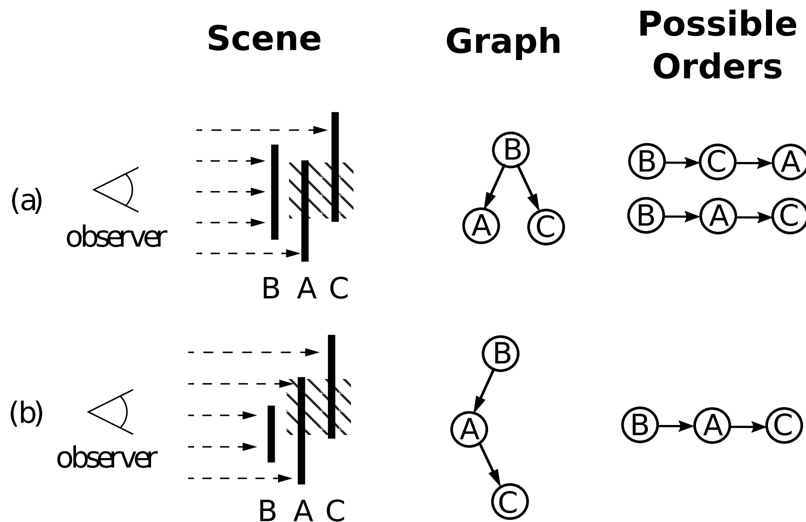


Figure 4.7: Two example cases where object *B* sits in front of *A* and *C*. The shaded region highlights the overlap between *A* and *C*.

It is easy to prove that if two nodes are not connected, then either they do not overlap at all, or their overlapping region is not visible, i.e., the missing relationship is not relevant in terms of

overshading. Figure 4.7 illustrates this property with two examples. In both cases the rendering order is A, B, C. In case (a), nodes A, C are not connected and therefore their overlapping area is not visible, so the different possible rendering orders between A and C do not produce a different amount of overshading. In case (b), the overlapping between A and C is partially visible and therefore these nodes are connected in the graph. That is, the partial order represented in the Visibility Graph contains all the precedence relations necessary to create an order where visible objects are scheduled before the ones that they occlude.

4.2.6 Visibility Rendering Order Adjustments

The Visibility Rendering Order that the Tile Scheduler receives from the Visibility Sort unit contains the object-ids of the objects rendered in the previous frame, and they may differ slightly from the objects to be rendered in the current frame. On the one hand, objects rendered in the previous frame are present in the Visibility Graph, but they are not present in the current frame and therefore they must not be scheduled, so they are simply discarded by the Tile Scheduler in a sequential pass. On the other hand, objects not present in the Visibility Graph but present in the new frame must be scheduled, so they are put in the list after the objects in the graph.

Note that objects with Depth test disabled or with Blending enabled cannot be simply put at the end of the order list because it could produce erroneous images. These objects must be scheduled in the same relative order as they appear in the program rendering order. VRO respects the OpenGL standard in the sense that the result is the same as if objects were processed in program rendering order, so this constraints are taken into account when creating the final rendering order. Fortunately, objects with Blending enabled or with Depth Test disabled are objects commonly part of the GUI of the applications and tend to be the last objects to appear in the program rendering order, so in the practice they introduce minor constraints to the Visibility Rendering Order.

4.3 Microarchitecture

This Section describes the implementation details of our technique on a contemporary GPU. Next, we describe the extensions to the baseline architecture that are required to support Deferred Rendering (DR) and our technique (VRO). DR and TBR will be used for comparison purposes in our experiments.

4.3.1 Deferred Rendering TBR GPU

Deferred Rendering (DR) is the state-of-the-art regarding overshading reduction, so we decided to model it for comparison purposes. As outlined in Section 4.1, DR reduces overshading by first performing Hidden Surface Removal (HSR), which in first place computes the final state of the Depth Buffer for a given tile. Thereafter, it starts an ordinary rendering of the tile. Hence, given that the Depth Buffer contains the depth of the visible objects, the Early-depth is able to discard all the occluded fragments and achieve minimum overshading.

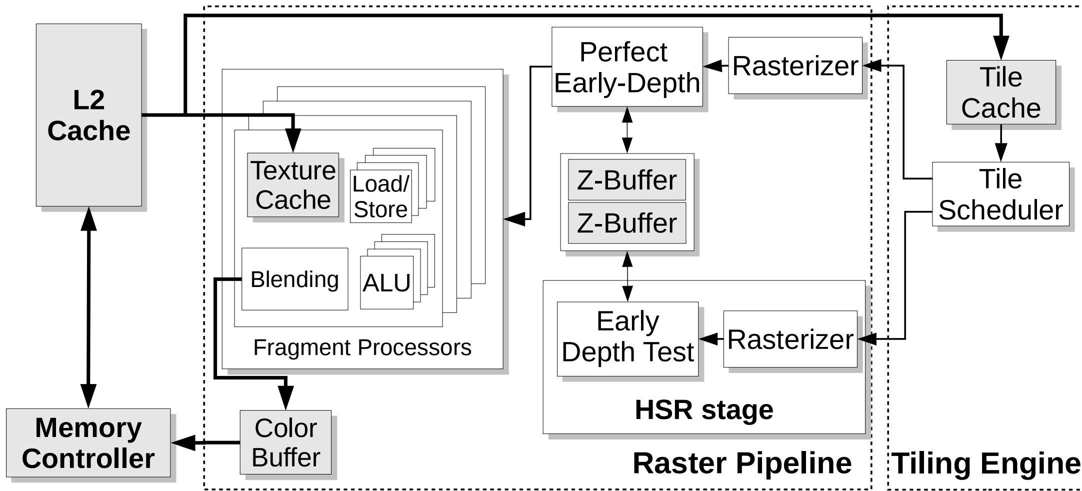


Figure 4.8: Raster Pipeline of a TBR GPU implementing Deferred Rendering.

A DR technique has been commercially implemented by Imagination Technologies in their tile-based PowerVR GPU family [91], which they refer to as a TBDR. However, since only partial information about this technique has been disclosed, our Deferred Rendering implementation models what we believe is the most optimistic interpretation of this partial information, in order to be used in the comparisons with our proposal.

As previously shown in Figure 4.3, we developed different implementations of DR: sequential DR (a) and parallel DR (b). Sequential DR is a naïve implementation that stalls the rest of the Raster Pipeline while performing HSR. The sequential implementation badly hurts both performance and energy compared with the baseline GPU. For this scheme, the execution time increases for every one of the benchmarks tested, by 23% on average. Regarding energy consumption, it increases around 6% on average when compared with the baseline GPU. These huge overheads are due to the fact that the total time of the HSR stage (only depth rasterization plus depth test) greatly exceeds the savings provided by the overshading reduction.

Nevertheless, these huge overheads can be removed by performing the HSR stage in parallel with the other stages of the Raster Pipeline (see Figure 4.8). Thus, in this optimized scheme (parallel DR), while the HSR is being executed for tile *i+1*, the rest of the Raster Pipeline is executed in parallel to render the tile *i*. Obviously, this parallel implementation introduces a hardware cost and some hardware blocks, such as the Rasterizer, the Early-depth test and the Depth Buffer, need to be replicated. Furthermore, the Tile Scheduler is equipped to handle memory requests of two primitives in parallel: one primitive from the tile being rendered and the other one from the tile in the HSR stage. This does not mean that the Tile Cache has now two read ports, but the Tile Scheduler will arbitrate between both request queues and only one will be sent to the Tile Cache each cycle in a Round Robin fashion. Even though this parallel implementation of DR introduces a non negligible amount of extra hardware (6% area overhead w.r.t baseline GPU), it outperforms sequential DR in both performance and energy, so it is the one we use in the results section to be compared against our technique, VRO.

4.3.2 Visibility Rendering Order TBR GPU

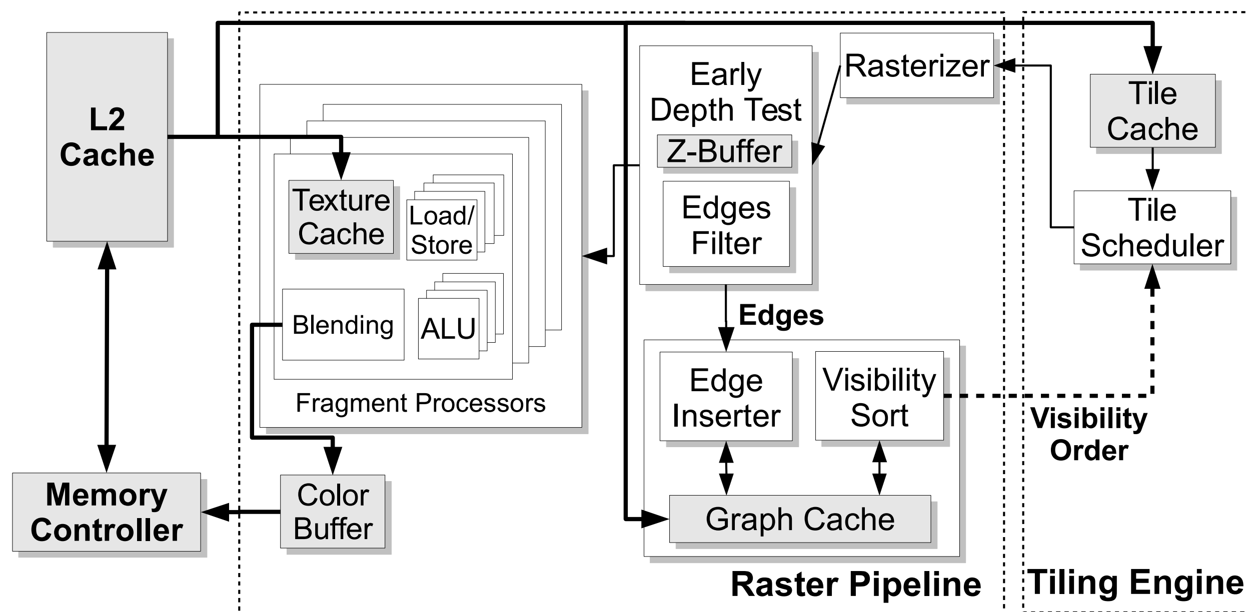


Figure 4.9: Raster Pipeline of a TBR GPU implementing VRO.

As Figure 4.9 shows, our technique includes several new pieces of hardware: the Edge Insertion unit, the Edge Filter, the Graph Cache and the Visibility Sort unit. As usual, the control of the new hardware has been implemented using FSMs.

On the one hand, the Early-depth unit sends the edges to the Edge Inserter unit, that stores them in the Graph Buffer. The Graph Buffer contains the Visibility Graph of the currently rendered frame. It is held in main memory and accessed through the Graph Cache. Edge insertions take place at fragment granularity using the results of Early-Depth comparisons. However, since graph edges represent object pairs there is a large amount of tests that actually produce the same edges. The Edge Filter is a small and fast associative on-chip structure that caches the most recently inserted edges and filters out 99.9% redundant insertions to the Graph Buffer, thus avoiding Graph Cache accesses. Thanks to this structure, the Edge Insertion unit accesses the Graph Cache on average much less than once every thousand fragments.

On the other hand, the Visibility Sort unit sorts the Visibility Graph and creates a preliminary ordered list of nodes, which is sent to the Tile Scheduler. This process is performed in parallel with the Geometry Pipeline execution of the following frame. We have measured that, for the worst case (*Forest 2*), the total time required for this process is almost two orders of magnitude smaller than the execution time of the Geometry Pipeline, so tiny overheads in execution time are introduced.

The Tile Engine traverses the list that contains the Visibility Rendering Order and updates it. Objects present in the previous frame but not present in the current frame are simply discarded by the Tile Scheduler. Objects not present in the previous frame but present in the current one must be

scheduled so they are added to the Visibility Rendering Order. After the adequate adjustments to satisfy the restrictions presented in subsection 4.2.6, the Tile Scheduler produces the final Visibility Rendering Order.

The mechanism of fetching the primitives from main memory is as follows. For a given tile, the Tiling Engine maintains a list (*Primitive List*) of pointers to the memory positions in the Parameter Buffer where the primitives of the tile are stored.

The Tile Scheduler reads the pointers of the *Primitive List* and then fetches the associated primitive data through the Tile Cache. In VRO (see Figure 4.10), the Tiling Engine includes another list (*First Primitive List*) that includes the position of *Primitive List* that points to the first primitive of an object in the given tile, as well as the number of primitives of such object in the tile. The mechanism of fetching the primitives in VRO is as follows. For every *Object-ID* in the Visibility Rendering Order ① (e.g. object D), VRO reads from *First Primitive List* ② the position of *Primitive List* ③ that corresponds to this object (e.g. position 6 for object D). This position holds the pointer to the first primitive of the object. Then, VRO fetches the data of the primitive from the *Parameter Buffer* ④. Given that *Primitive List* is located in consecutive positions of main memory, VRO reads subsequent pointers to primitives from *Primitive List* in the same manner as the baseline does.

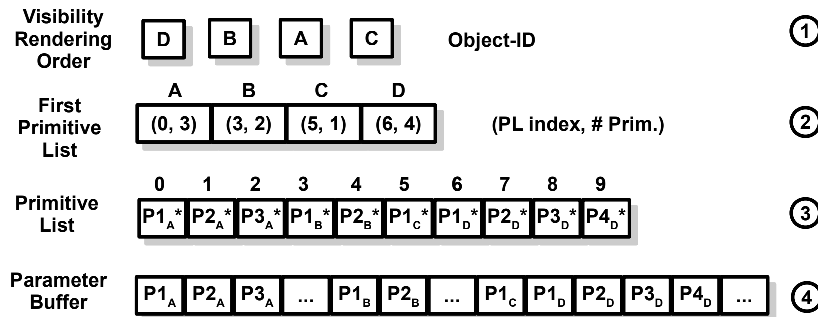


Figure 4.10: Detail of Tile Engine structures involved in Geometry Fetching.

Like in the baseline GPU, once the Geometry Pipeline has been executed, the Raster Pipeline renders the frame tile by tile. However, instead of reading the primitives in program rendering order the Tile Scheduler reads the primitives in Visibility Rendering Order. VRO increases the culling effectiveness of the Early-depth test and reduces overshading, which decreases the total number of instructions and texture accesses executed in the Fragment Processors. In order to do a fair comparison between VRO and DR, the Tile Scheduler of VRO is equipped with the same hardware that is included in DR to handle memory requests of two primitives in parallel.

Graph Buffer

The Graph Buffer is a small array in system memory where the Visibility Graph is stored. Due to the fact that the Visibility Graph is very sparse, it is represented as a set of adjacency lists, one per node. Each adjacency list is implemented as a linked list of one or more entries. Each entry contains the object-id of up to W children nodes as well as other metadata shown in Figure 4.11.

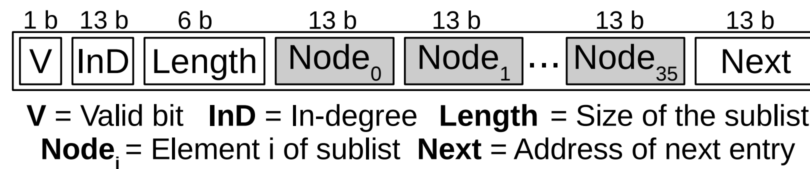


Figure 4.11: Detail of an entry of the Graph Buffer. Each entry is 512 bits (including 11 bits of padding).

VRO performs operations like membership and insertion in one step whenever the adjacency list contains less than W children nodes. Furthermore, our scheme is not constrained to a maximum of W outgoing edges per node. The adjacency lists are extended dynamically to any number of edges by allocating one or more extra overflow entries of the Graph Buffer if required. Primary lists are sequentially allocated from the lowest addresses of the buffer onwards and overflow lists are allocated from the highest addresses backwards. A buffer with N entries can store up to N nodes if none of their lists overflow, but most importantly, it can contain lists with theoretically unlimited number of edges per node.

Of course, there is a limitation imposed by the size of the memory region devoted to the Graph Buffer. However, we show that the size of the Graph Buffer represents a small region of main memory. For example, with 8192 entries and 64 B per entry the Graph Buffer is 512 KB. Figure 4.12 plots the amount of main memory to be allocated to the Graph Buffer for different number of maximum objects (nodes) and different number of children nodes per entry of the buffer (W). Note that even for a number of objects three orders of magnitude higher than the average number of objects observed in our set of benchmarks the memory region devoted to contain the Graph Buffer would be smaller than 11 MB. Although all our benchmarks have less than 256 nodes (see Top part of Figure 4.13) we provision for a much larger number of objects, so the graph has been sized to 8192 entries which is more than enough to support common mobile workloads. Note that an object corresponds with a 3D model composed of different primitives and not a single one.

There exists a clear trade-off in the implementation of the structure of the Graph Buffer. The smaller the number of children nodes in one entry of the buffer, the smaller the total size of the Graph Buffer, but the higher the number of cycles to read the whole adjacency list of a node if it contains more than W children nodes. Figure 4.13 (Bottom) plots the 75th, 85th and 95th percentiles of the largest adjacency lists sizes, and it shows that most of them contain a small number of nodes (objects). For example, in the case of *300*, the values for P75, P85, and P95 mean that the 75%, 85%, and 95% of the lists contain less than 12, 17, and 27 children nodes respectively.

In the worst case, around 95% of adjacency lists of the Visibility Graph of our benchmarks have 35 or less edges per node. Hence, we allocate 36 edges per entry (W = 36), and in this way, the size of one entry is slightly smaller than 64 B and fits into a single cache block. Accounting for 8192 entries and 64 B per entry, the total size of the Graph Buffer in main memory is 512 KB. However, nothing impedes reserving more main memory to provision for a larger number of objects. In any case, in order to reduce main memory traffic and latency, the access to the Graph Buffer is done through the Graph Cache, which is 4 KB 4-way associative (92.6% hit ratio on average).

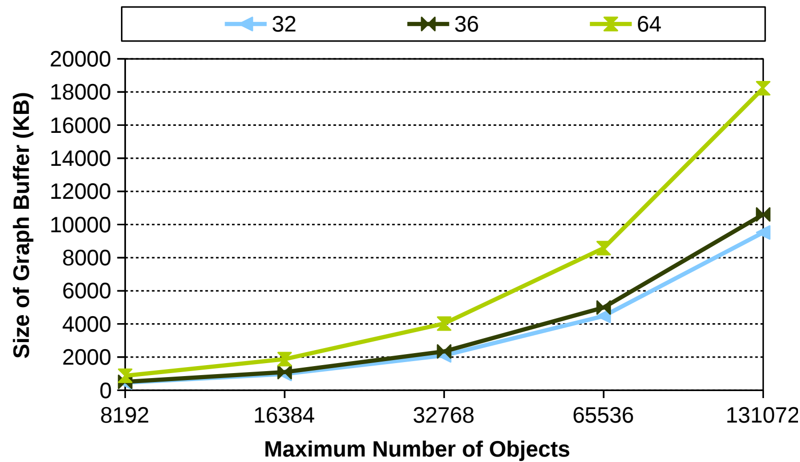


Figure 4.12: Size of the Graph Buffer for different number of children nodes (W) and different number of maximum objects (from 8192 to 131072).

Edge Inserter

The Edge Inserter is the unit responsible for creating the Visibility Graph. It works in parallel with the other stages of the Raster Pipeline. It not only creates the Visibility Graph, but also computes the in-degree of each node (see the field InD in Figure 4.11), which will be used by the Visibility Sort unit. The Edge Inserter (see Figure 4.14) receives through a queue the edges from the Early-depth test which were not filtered by the Edge Filter. The insertion of an edge (N_{src}, N_{dst}) in the Visibility Graph is a three-step process (see Figure 4.14):

1. The primary entry of node N_{src} is read from the Graph Buffer to the AdjacencyList-Reg.

2. Then the AdjacencyList-Reg is searched to check if the edge was previously inserted in the Visibility Graph. If the primary list of node N_{src} has no linked entries, this can be done in a single clock cycle with an array of equality comparators. Otherwise, one or more overflow entries may be subsequently read and copied to the AdjacencyList-Reg until the edge is found or the last entry is read.

   (a) If the edge already exists, then it is discarded.

   (b) Otherwise, the *Length* field is increased and the edge is added to the adjacency list. However, if the entry is full, a free overflow entry is first assigned to node N_{src} and its address is stored into the *Next* field and written back to the Graph Buffer.

3. Finally, if a new edge has been added to the graph, the in-degree of the target node N_{dst} is increased.

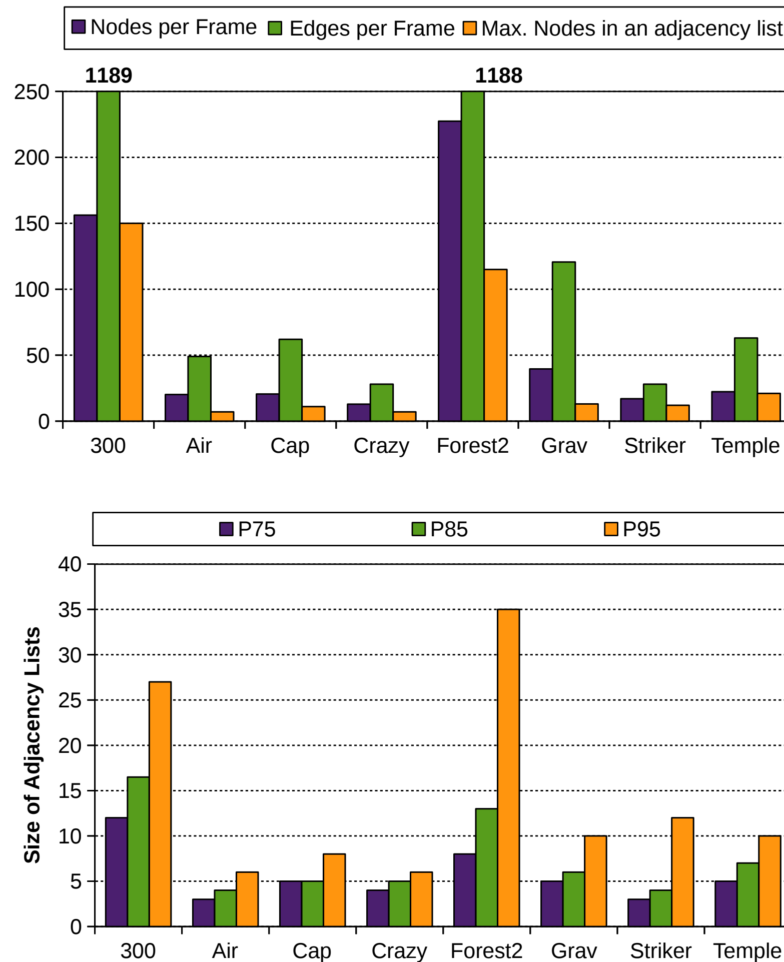


Figure 4.13: (Top) Nodes per frame, edges per frame and maximum number of nodes in an adjacency list. (Bottom) 75th, 85th and 95th percentiles of the size of the adjacency-lists of the scene graphs analyzed.

Visibility Sort

The Visibility Sort unit is responsible for sorting the Visibility Graph. As outlined above, it implements an extended version of Kahn's algorithm able to handle cycles. The unit works in two phases (see Figure 4.15):

1. **Initial search.** The primary entries of all the nodes in the Visibility Graph are read from the Graph Buffer. If the node is a root (InD $= 0$), its object-id is pushed into the Roots Queue.
2. **Iterative procedure.** The object-ids stored in the Roots Queue are iteratively processed. For each node in the queue:
   (a) The adjacency list of the node is read from the Graph Buffer into the AdjacencyList-Reg. At the same time, the object-id is sent to the Tile Scheduler through the Order Queue.

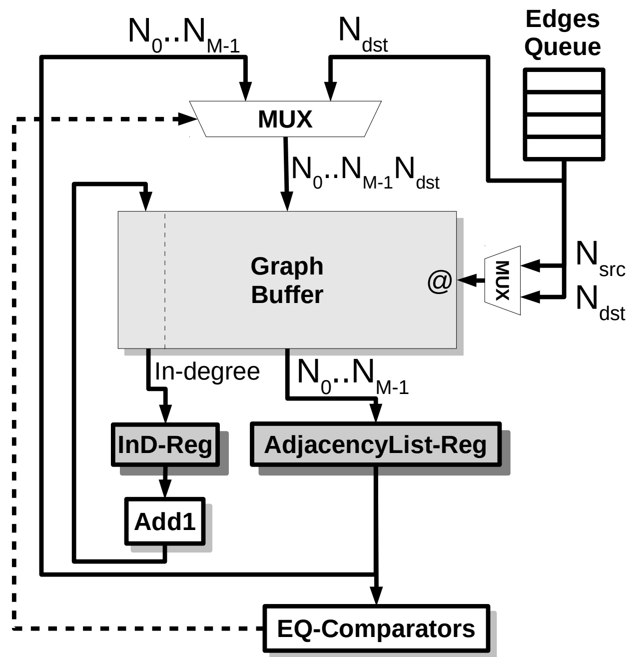


Figure 4.14: Edge Insertion Hardware.

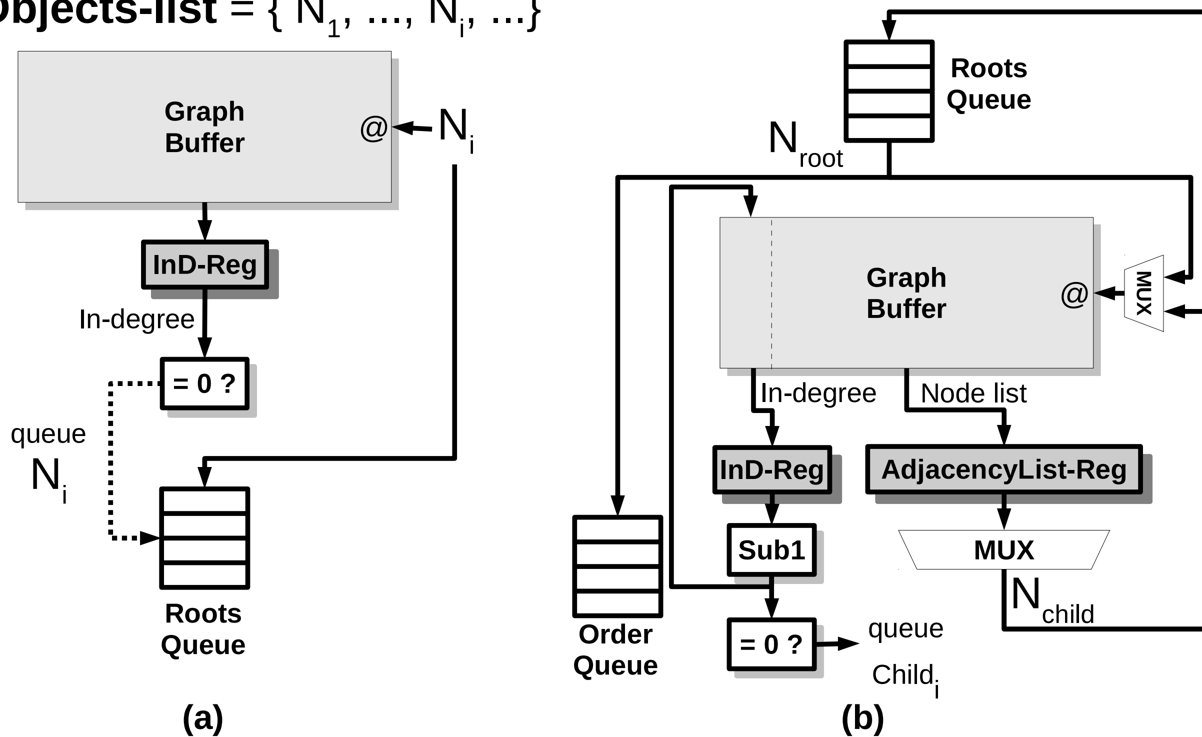


Figure 4.15: Visibility Sort Hardware. (a) Initial search (b) Iterative procedure

(b) For each child in the AdjacencyList-Reg, the in-degree is read from the Graph Buffer, and if it is still a valid node, it is decremented and written back. If the in-degree of a child becomes zero (becomes a root), it is pushed into the Roots Queue. In case of an adjacency list with overflow entries they are read in turn and processed in the same way.

(c) The node is invalidated.

If the Roots Queue becomes empty and all the graph nodes have been sorted, the algorithm finishes. Otherwise, a cycle has been found, so the unit traverses the nodes in program order and selects the one with minimum in-degree (min-comparator not shown in Figure 4.15 for the sake of clarity), pushes it into the Roots Queue, and then resumes the iterative procedure. Note that the node has incoming edges remaining in the adjacency lists of its ancestors. However, since the node is invalidated after being processed, its in-degree will never be decremented again.

Identification of Objects

Object identifiers are required by the Visibility hardware unit to identify the objects across different frames. We need, therefore, to maintain the object identifier for objects along the graphics pipeline up to the Early-depth (that is extended with object ids) and the VRO unit.

A simple way to do this is to include an object identifier in every draw command of an object. This could be done using the debug marker extension of OpenGL [191], implemented in OpenGL ES 1.1 and 2.0. This extension allows the programmer of an application to annotate the OpenGL command stream with a descriptive text marker. This extension relies on the driver and the hardware to maintain the object notion through the rest of the graphics pipeline. Note that current 3D applications already uniquely identify the objects of the scene [144], so the requirement here is to pass this information from the application layer to the GPU.

4.4 Experimental Framework

In our experiments, we use the Teapot simulation framework [111]. We model not only the baseline GPU architecture, which closely resembles that of the Utgard microarchitecture of ARM Mali [89], but also we model Deferred Rendering and Visibility Rendering Order techniques on a TBR GPU architecture. ARM Mali Utgard microarchitecture is the most successful mobile GPU till the date, with around 19.1% of the mobile GPU market share by March 2017 [11], while TBR GPUs represent around 95% of the mobile GPU market. Although in this work we employ a TBR GPU architecture, note that VRO is orthogonal to the TBR mode, and its implementation on top of an IMR GPU would also increase the performance and reduce the energy consumption of the GPU as we show in Appendix A.

Table 4.2: GPU Simulation Parameters.

Baseline GPU Parameters	
Tech Specs	400 MHz, 1 V, 32 nm
Screen Resolution	1200x768
Tile Size	16x16
Queues	
Vertex (2x)	16 entries, 136 bytes/entry
Triangle, Tile	16 entries, 388 bytes/entry
Fragment	64 entries, 233 bytes/entry
Caches	
Vertex Cache	64 bytes/line, 2-way associative, 4 KB, 1 bank, 1 cycle
Texture Caches (4x)	64 bytes/line, 2-way associative, 8 KB, 1 bank, 1 cycle
Tile Cache	64 bytes/line, 8-way associative, 128 KB, 8 banks, 1 cycle
L2 Cache	64 bytes/line, 8-way associative, 256 KB, 8 banks, 2 cycles
Color Buffer	64 bytes/line, 1-way associative, 1 KB, 1 bank, 1 cycle
Depth Buffer	64 bytes/line, 1-way associative, 1 KB, 1 bank, 1 cycle
Non-programmable stages	
Primitive assembly	1 triangle/cycle
Rasterizer	4 attributes/cycle
Early Z test	32 in-flight quad-fragments, 1 Depth Buffer
Programmable stages	
Vertex Processor	1 vertex processor
Fragment Processor	4 fragment processors
Latency Main memory	50-100 cycles
Bandwidth	4 bytes/cycle (dual channel)
Extra Hardware VRO GPU	
Edges Filter	32 elements, LRU, 1 cycle
Graph Cache	64 bytes/line, 4-way associative, 4 KB, 1 bank, 1 cycle
Edge Insertion	1 Edge Inserter unit
Graph Sort	1 Visibility Sort unit
Edges Queue	64 entries, 4 bytes/entry
Order Queue	64 entries, 2 bytes/entry
Extra Hardware DR HSR stage	
Tile Queue	16 entries, 388 bytes/entry
Fragment Queue	64 entries, 233 bytes/entry
Rasterizer	4 attributes/cycle
Early Z test	32 in-flight quad-fragments, 1 Depth Buffer
Depth Buffer	64 bytes/line, 1-way associative, 1 KB, 1 bank, 1 cycle

4.4.1 GPU Simulation

Teapot [111] is a mobile GPU simulation infrastructure that can run unmodified commercial Android applications. It includes an OpenGL commands interceptor, a GPU trace generator and a cycle-accurate timing simulator. The parameters used in the simulations are shown in Table 4.2.

While a graphical application is executed in the Android emulator [87], a trace of OpenGL commands is stored. This trace is later fed to the GPU trace generator, which creates the GPU trace through the software renderer (Softpipe) included in Gallium3D [88]. The generated GPU trace file includes the Vertex Processor and Fragment Processor instructions, the memory addresses of the texture and vertex data, the primitives generated, and the corresponding fragments as well as other pipeline data required to simulate the execution. The GPU trace is fed to the cycle-accurate timing simulator, which accurately models the baseline GPU. This simulator has been extended to implement both DR and VRO GPUs as described in Section 4.3.

The results reported include static and dynamic energy consumption of the whole GPU, including RTL models of Edge-Insertion and Visibility-Sort, as well as the full memory hierarchy including the main memory. Teapot models the power of the GPU with McPAT [163]. Likewise, the power of the VRO unit has been modeled using McPAT's components, shown between parenthesis in the following list: Graph Cache (Cache); EQ Comparators (XOR); Muxes (MUX); Min-Comparator (ALU); Adders (ALU); Subtractors (ALU); and registers. The area overhead of VRO is less than 1% whereas for DR it is around 6% (with respect to baseline TBR in both cases).

4.5 Experimental Results

In this section we present the performance and energy savings of VRO with respect to the baseline TBR GPU. Furthermore, the benefits of VRO are compared with those of DR.

4.5.1 Effectiveness of VRO

Figure 4.16 shows the normalized speed-up achieved by our technique (VRO) and by DR relative to the ARM Mali-like baseline TBR GPU. As it can be observed, VRO achieves up to 1.42x speed-up (*Forest 2*), and 1.27x on average, being the lowest speed-up 1.14x (*Captain America*). DR achieves up to 1.25x speed-up (*Striker*), and 1.17x on average, being the lowest speed-up 1.13x (*Gravity*). Regarding system energy (see Figure 4.17), the consumption of VRO is reduced up to 0.76x (*Forest 2*) and 0.84x on average, being the lowest reduction 0.91x (*Captain America*). DR reduces it up to 0.82x (*Forest 2*) and 0.88x on average, being the lowest reduction 0.96x (*300*). Recall that for sequential DR (not included in the graph), the execution time increases for every one of the benchmarks tested, 23% on average, while the energy consumption increases around 6% on average when compared with the baseline GPU.

The performance advantage of VRO with respect to DR is the result of several factors. On the one hand, DR may reduce more overshading than VRO because it works at pixel granularity whereas VRO reorders the geometry at object granularity. The extra fragments processed by VRO

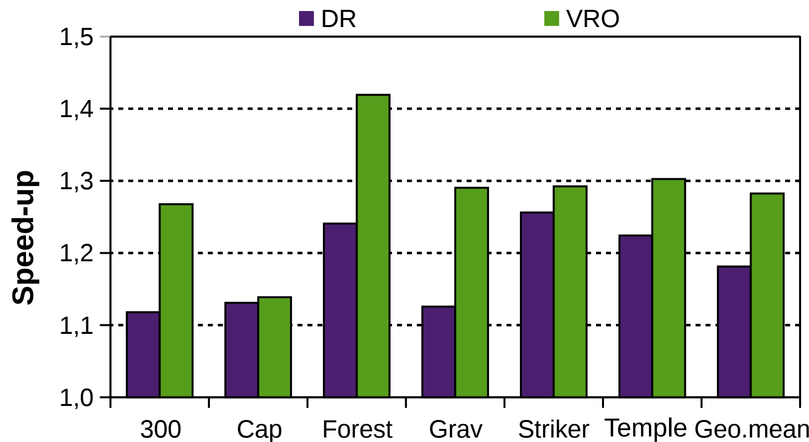


Figure 4.16: Speed-up of DR and VRO normalized to the baseline TBR GPU.

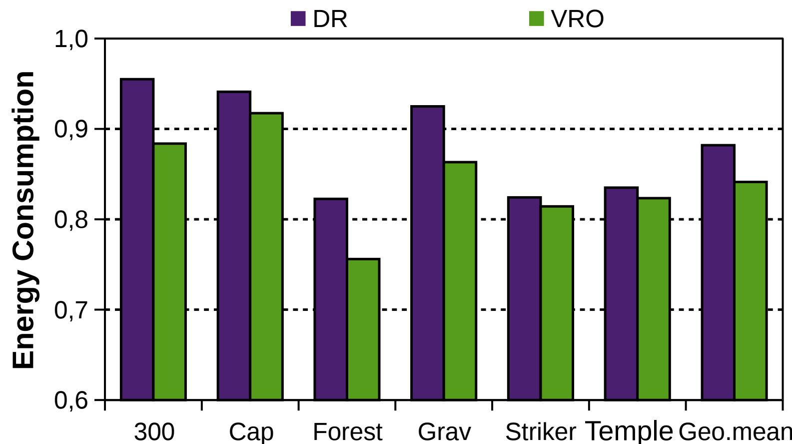


Figure 4.17: Energy consumption of DR and VRO normalized to the baseline TBR GPU.

may induce pipeline stalls and hurt performance only in case that they fill the queue that feeds the Fragment Processors. On the other hand, the Tile Scheduler of DR reads in parallel primitives of tile i+1 for the HSR unit, and primitives of tile i for the conventional Raster Pipeline, which may increase latency and starve the Fragment Processors. For an equally sized available Tile Cache bandwidth, DR produces substantially more accesses to the cache (which may degrade throughput) and has a larger working set (which may degrade miss rate and latency). To show the relative importance of these factors, we have measured both the overshading and the average fetch time to the Tile Cache.

Figure 4.18 plots the overshading for DR and VRO normalized to the overshading of the baseline GPU. As expected, the overshading with DR (close to 0.7x) is smaller than the overshading with

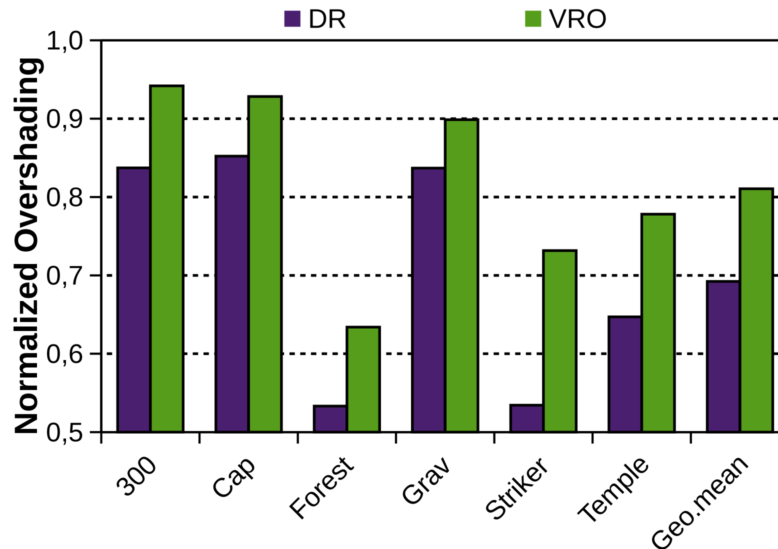


Figure 4.18: Overshading of DR and VRO normalized to the overshading of the baseline TBR GPU.

VRO (close to 0.81x). The smaller overshading reduction of VRO is mainly caused by the fact that DR performs the HSR stage at pixel level granularity, while VRO performs the sorting at object-level granularity.

Figure 4.19 plots the average fetch time per primitive in cycles for DR and VRO. It shows substantial fetch time increases for DR with respect to VRO. On average, the fetch time for DR is 68 cycles while it is only 50 cycles for VRO (25% less). Furthermore, the number of primary misses of the Tile Cache is 11% higher for DR than for VRO.

Figure 4.20, shows the normalized memory-traffic of VRO and DR with respect to baseline GPU. As can be seen the bandwidth of VRO and DR is 0.98 and 0.96 respectively. On the one hand, because of its lower overshading, DR saves more texture traffic than VRO. But on the other hand, DR must read twice the number of primitives to execute the HSR phase, which increases main memory traffic. Regarding the extra accesses of VRO to the Visibility-Graph, they add less than a tiny 0.005% to the total memory traffic.

Figure 4.21 compares the relative importance of the above two factors and explains why VRO outperforms DR. It plots the absolute time difference in cycles of DR with respect to VRO, for different parameters: total number of cycles to fetch primitives (first bar), total number of pipeline stall cycles caused by Fragment Processor input queue full (second bar), and total execution time (third bar). The first bar shows that DR spends many more cycles than VRO to fetch primitives, more than 27 Million cycles on average, which is caused by the higher latencies reported in Figure 4.19. The second bar shows that DR experiences less stall cycles caused by busy Fragment Processors, about 2.85 Million cycles less than VRO. This is related to the better overshading reduction of DR reported in Figure 4.18. Note however that not all the extra fragments of VRO cause a pipeline

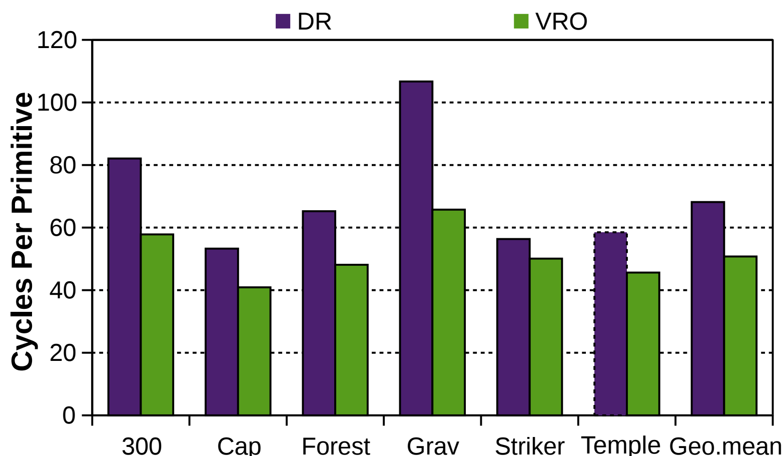


Figure 4.19: Number of cycles to read a primitive with DR and VRO.

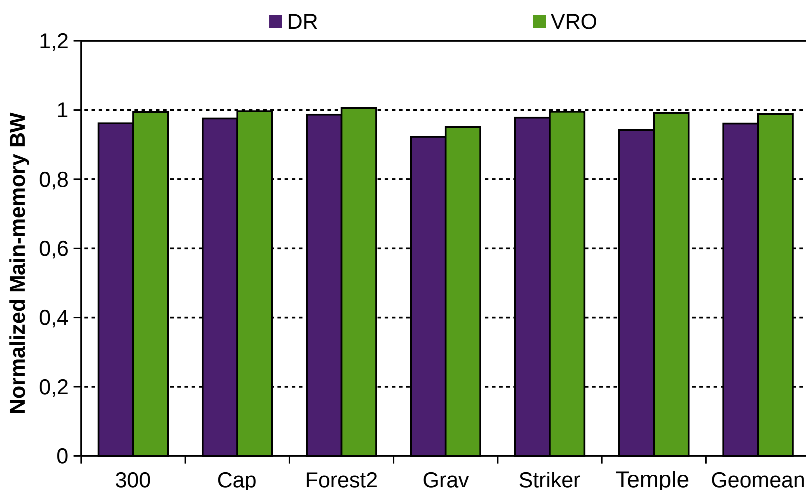


Figure 4.20: Normalized memory traffic of DR and VRO with respect to baseline GPU.

stall, only in case that they fill the queue that feeds the Fragment Processors.

The third bar is not just the sum of the other two factors. Not all the extra fetch cycles incurred by DR are ultimately translated to net increases of the execution time, because the buffers in between the Raster Pipeline stages partially smooth the effect of the initial fetching overheads. The increment in fetching time is ultimately translated to an increment in execution time around 21.6 Million cycles on average, as expected from the large difference between the other two bars. It explains the speed-ups reported in Figure 4.16.

In benchmarks such as *300*, *Forest 2* and *Gravity* the extra fetching cycles are largely translated into extra execution time of the Raster Pipeline. This is because these benchmarks have around

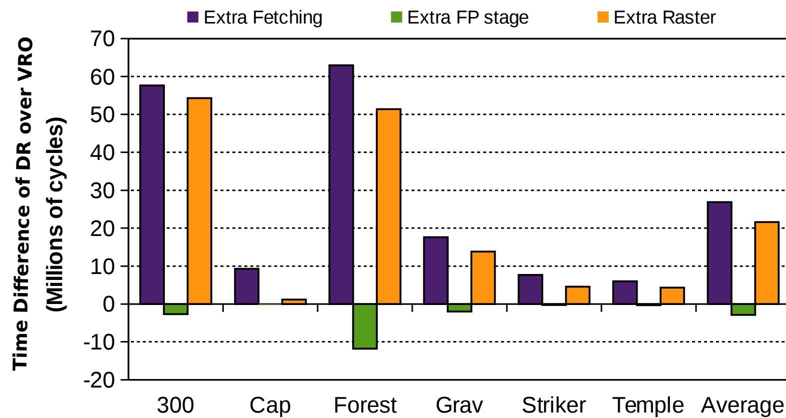


Figure 4.21: Increment of cycles reading geometry (first bar), increment of stall cycles caused by the Fragment Processing stage (second bar), and increment of cycles of execution of the Raster Pipeline (third bar) all using DR with respect to VRO.

one order of magnitude more primitives than the other ones, which means that the overhead of reading the geometry relative to the total time of the Raster Pipeline is greater than in the other benchmarks. Furthermore, these benchmarks have less fragments per primitive than the others. The smaller the number of fragments per primitive the faster the queue that feeds the Rasterizer gets empty. In the case of *Captain America*, the initial increment of the fetching cycles is hardly reflected as an overhead in the total processing time of the Raster Pipeline. Unlike other benchmarks (*300*, *Forest 2* and *Gravity*), *Captain America* has a much lower number of primitives and a greater number of fragments per primitive.

Figure 4.22 shows the normalized energy breakdown for both DR and VRO. The energy advantage of VRO with respect to DR is the result of two main factors. The most important one is the greater speed-up of VRO (0.1x greater than DR's speed-up on average), which is translated into a smaller static energy for the Main-memory/GPU system. Although DR reduces more overshading than VRO, and therefore reduces the dynamic energy of the fragment processors and the main memory among others, VRO is able to compensate it with a significant reduction in static energy. On the other hand, DR exhibits greater raster activity than VRO because it performs the HSR stage to cull hidden fragments, which requires reading and rasterizing the primitives of the scene two times (only position).

In conclusion, we have shown that even reducing less overshading than DR, VRO achieves higher speed-up because the overhead in fetch cycles with DR is much higher than the overhead caused by the extra fragment processing with VRO. Moreover, take into account that the area overhead of VRO is less than 1% whereas the area overhead of DR is around 6%. Therefore, some of this area could be used to implement more complex schemes of VRO in order to further reduce overshading. The overshading can be differentiated in two types: intra-object and inter-object overshading. The former is produced by auto-occlusions of an object. The latter is the overshading caused by occlusions between different objects. Given that VRO sorts at object-level granularity, it is only reducing inter-object overshading. However, as we show in Section 1.4.1, there are techniques

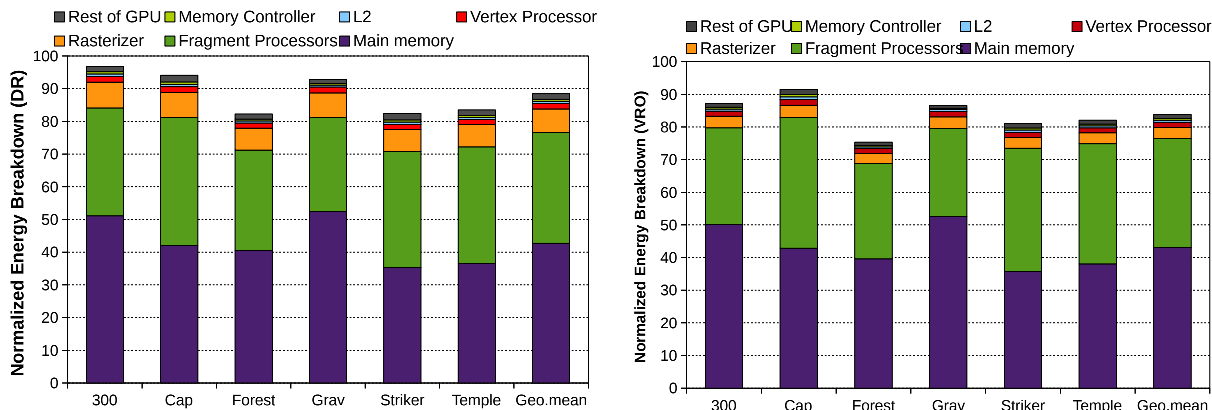


Figure 4.22: Energy breakdown for the system Main-Memory/GPU with DR (left) and VRO (right) both normalized to the baseline GPU.

that are complementary to VRO and that effectively reduce the intra-object overshading. Hence, we believe that VRO has still room for improving performance and energy savings by combining it with one of those techniques. Likewise, in Section 6.2 we introduce a possible extension to VRO that aims at reducing intra-object overshading.

4.5.2 Overshading with Different Heuristics to Break Graph Cycles

This subsection analyzes the overshading reduction effectiveness of the different heuristics to break graph cycles that were previously introduced in subsection 4.2.4. The main goal of VRO is to reduce the overshading by culling hidden surfaces before they reach the Fragment Processing stage, so that the number of instructions executed in the Fragment Processors is reduced. As can be seen in Figure 4.23, *C_H1* almost obtains the same overshading reduction of *Perfect Object Order* (obtained with our greedy algorithm). If we compare both alternatives of *H1* with both alternatives of *H2*, when a graph cycle is found, *H2* needs to find the object with minimum average depth, thus *H2* must compute the average depth of all objects. On the other hand, *H1* only needs to find the node with minimum in-degree. In short, *H1* not only performs better than *H2* but also has a smaller implementation cost. Regarding the comparison between *C_H1* and *PO_H1*, although *C_H1* gets impressive results, its implementation cost is higher. *C_H1* needs extra memory space to store the counters (as well as hardware to compute them). The value of these counters can be very high because we are counting the number of times an object is occluding another object, which may happen millions of times at fragment granularity. Furthermore, when creating the Visibility Order, *C_H1* needs to compare these counters for each pair of parallel edges to know which object goes first. We also test *MinZ*, an heuristic that selects the next node with minimum average depth when a cycle is found when traversing the graph. As can be seen, this alternative achieves the worst results among all the tested heuristics.

We decided to implement *PO_H1* despite it is the second scheme in overshading reduction because we took into account not only the overshading reduction but also the complexity of the implementation and its associated cost. Note that, regardless of the approximation applied in order

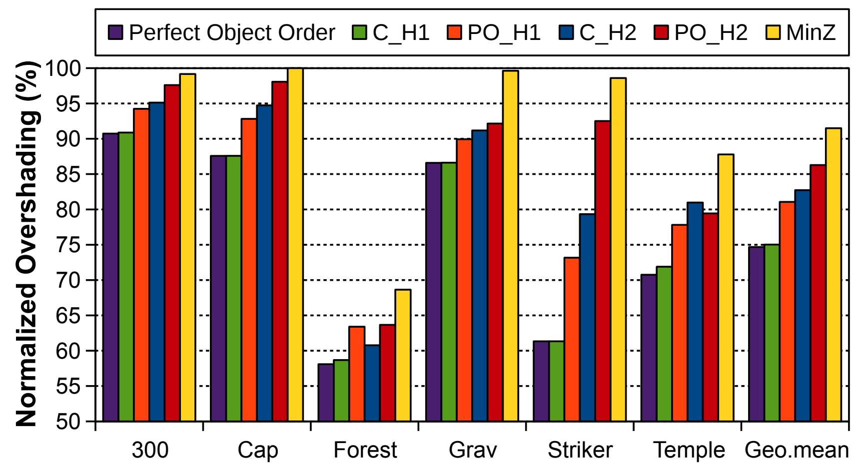


Figure 4.23: Normalized Overshading with different Heuristics to break Graph Cycles.

to achieve a cost effective heuristic correctness is guaranteed. Besides, as observed in Figure 4.23, for all the different versions of VRO the overshading is reduced.

4.6 Conclusion

In this chapter we have presented VRO, a novel technique that effectively reduces overshading. VRO is based on the observation that the relative order among the objects of a scene tends to be very similar between one frame and the next. VRO includes a small hardware unit that stores the order relations among the objects of a scene of the current frame in a buffer. This information is used in the next frame, while the GPU is executing in parallel the Geometry Pipeline, to create a Visibility Rendering Order that guides the Tile Scheduler. The overhead of this technique 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.

For a set of unmodified commercial applications, VRO outperforms state-of-the-art techniques in performance and energy consumption by reducing the overshading without the need of an expensive HSR stage at fragment granularity. VRO is especially efficient for geometry-complex applications, which are expected to be the most common applications in mobile devices as they already are in desktops. VRO achieves a speed-up about 1.27x and an energy consumption around 0.85x compared to an ARM Mali-like GPU. VRO outperforms DR because the Visibility Rendering Order is created out of the critical path while DR introduces significant overheads to perform the HSR stage.

5

Render-Based Collision Detection for CPU/GPU Systems

Graphics animation applications such as 3D games represent a large percentage of downloaded applications for mobile devices and the trend is towards more complex and realistic scenes with accurate 3D physics simulations, like those in laptops and desktops. Collision detection (CD) is one of the main algorithms used in any physics kernel. However, real-time highly accurate CD is very expensive in terms of energy consumption and this parameter is of paramount importance for mobile devices since it has a direct effect on the autonomy of the system.

In this chapter, we present an energy-efficient, high-fidelity CD scheme that leverages some intermediate results of the rendering pipeline. It also adds a new and simple hardware block to the GPU pipeline that works in parallel with it and completes the remaining parts of the CD task with extremely low power consumption and higher performance than traditional schemes. Using commercial Android applications, we show that our scheme reduces the energy consumption of the CD by 99.8% (i.e., 448x times smaller) on average. Furthermore, the execution time required for CD in our scheme is almost three orders of magnitude smaller (600x speedup) than the time required by a conventional technique executed in a CPU. These dramatic benefits are accompanied by a higher fidelity CD analysis (i.e., with finer granularity), which improves the quality and realism of the application.

5.1 Collision Detection

During the past decade, mobile devices have quickly incorporated graphics and animation capabilities that in the 1980s were only seen in industrial flight simulators and a decade later became

popular in desktop computers and game consoles. The trend going forward is towards more powerful support for real-time physics simulations in all mobile devices, with increasing precision and realism.

Physics kernels contain several algorithms that manage the dynamics of the animations. CD is one of the most important algorithms since it identifies the contact points between the objects of a scene, and determines when they collide. It has been investigated for more than twenty years in different areas like computer graphics, robotics, or virtual reality.

There are many proposals to detect object interference through geometric computations, either by computing intersections between pairs of geometric primitives such as triangles or spheres, or by computing the distance between points in the 3D space. CD techniques are intrinsically quadratic with respect to the number of objects and their surfaces. To alleviate this cost, CD is often split into two steps: broad phase and narrow phase.

1. Broad Phase: A fast, simple, and often coarse grain test applied to all collisionable objects in a 3D Scene. It usually considers simple bounding boxes (often Axis Aligned Bounding Boxes, AABBs) around the objects. The pairs of objects whose bounding boxes collide are included in the Potentially Colliding Set (PCS).
2. Narrow Phase: Applied to all pairs of objects in the PCS. This test uses a more accurate shape of the objects to test the collision, and therefore it is more complex than the broad phase.

There exists a large body of research on CD [158, 196]. Both broad and narrow phases can be executed in a CPU or a GPGPU, depending on the characteristics of the specific platform. In most cases, the narrow phase of CD is executed in the CPU because of the non-regular nature of the computations, and in low-power systems the broad phase is executed in the CPU as well. On the other hand, there is a group of CD algorithms known as Image-Based CD (IBCD). They consist of the rasterization of the surfaces of the scene objects and the detection of their intersections based on the pixel depths of the corresponding fragments [115]. As shown by previous works (see Section 1.3.2), these kind of techniques have been proposed to exploit the computing power of graphics processors and their ability to rasterize polygons efficiently.

Broad phase algorithms are simple to parallelize, whereas narrow phase algorithms are usually, for a given pair of objects, control-intensive. However, on scenerarios where the limited energy consumption constrains the amount of computation (like in mobile devices), real-time CD is often restricted to a simple bounding volume model analysis. Render-Based Collision Detection (RBCD) addresses this problem by presenting a low-power yet detailed CD technique.

5.1.1 Image Based Collision Detection

The pioneering work of Shinya and Forgue [189] opened the path to perform CD in graphics hardware. This proposal consists of four steps. In the first step, the scene objects are projected onto a given plane in 3D space, for instance the screen plane, thus obtaining the coordinates (x,y) and the depth (z-value) of each vertex with respect to that plane. The second step consists of rasterizing

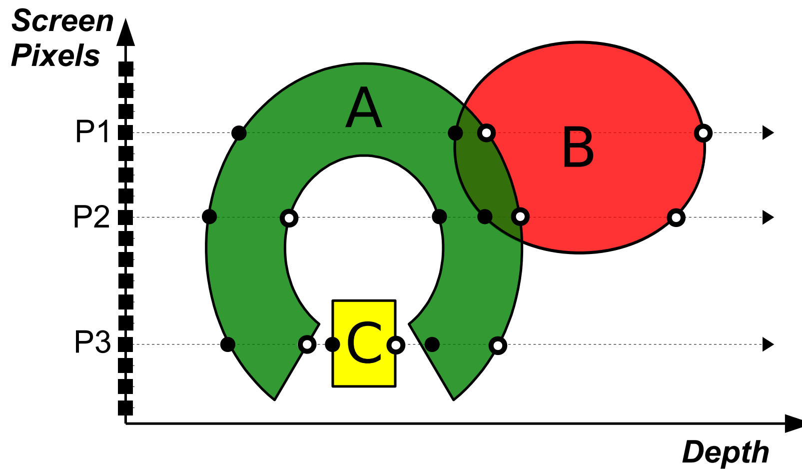


Figure 5.1: Discretized representation of the entry and the exit points of the surfaces in a 3D scene for pixels P1, P2 and P3. The Y-axis is a one-dimensional representation of the projection plane and the Z-axis represents depth.

all the surfaces of the collisionable objects, regardless of their visibility, both front and back faces. This process results in a collection of per-pixel lists of z-values, each indicating the depth of some point in one of the surfaces. Each list element also includes, along with the z-value, the object identifier. In the third step, for each pixel, the elements of the list are sorted by depth. In the fourth step, for each pixel, the algorithm detects possible overlaps between z-ranges of different objects.

Figure 5.1 illustrates an example with three objects A, B, C. Objects A and B are colliding, because their z-ranges overlap at pixels P1 (z11-z12) and P2 (z23-z24). However, there is no collision at pixel P3 because the z-ranges of the three objects are disjoint.

The main advantages of this algorithm are its simplicity, its time complexity proportional to the number of faces per object, the possibility of being accelerated with a GPU, and its applicability to any kind of renderizable surface. On the other hand, its resolution is finite (pixel resolution), but it has more than enough accuracy for practical purposes of visual realism in real time computer animation.

The scene shown in Figure 5.2 illustrates the accuracy of CD using AABBs (broad phase), the Gilbert-Johnson-Keerthi algorithm [140] (GJK, narrow phase) implemented in Bullet and RBCD. Due to the large false collisionable area that AABBs add to object A, false collisions are detected for pairs (A,C) and (A,B). With GJK, a false collision is still detected for pair (A,C). GJK only works with convex shapes, so if applied on a complex concave shape like A, one option is to use the convex hull of the shape, which results in adding a false collisionable area. In contrast, RBCD takes advantage of the discretized collision shape, which makes the false collisionable area much smaller than the other schemes, and thus does not produce any false collision for the given example. Note that the higher the rendering resolution, the smaller the false collisionable area that RBCD adds to the shapes of the collisionable objects.

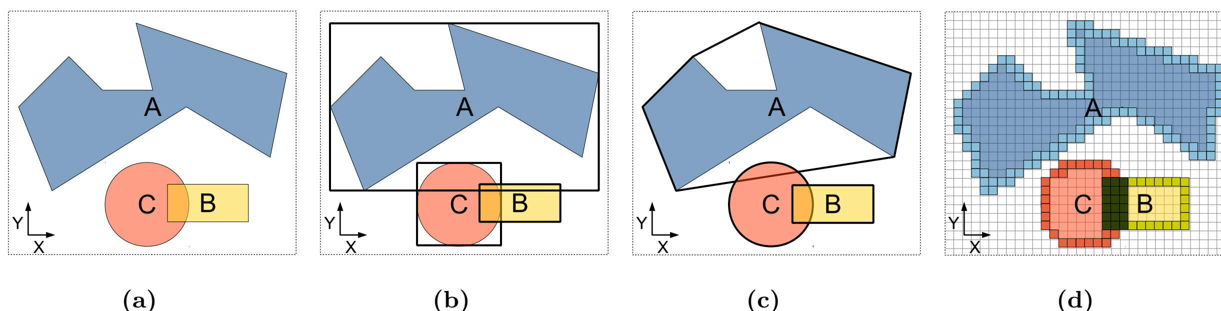


Figure 5.2: (a) Front view of a 3D scene (b) AABBs as collisionable shapes (c) Convex hull for GJK algorithm (d) RBCD.

5.1.2 Enabling RBCD in the GPU

The tasks involved in the first two IBCD steps, i.e. projection and rasterization, are already done in the image rendering of a 3D scene. Effectively, in a typical GPU the projection of the objects onto the screen is performed by the Geometry Pipeline and the rasterization occurs in the Raster Pipeline. Therefore, it would be possible to send some rendering results (the fragment screen coordinates and the per-pixel z-depth values) to the CPU, so that the remaining two IBCD steps (z-depth sorting and z-overlap analysis) would be performed by the CPU. However, the required communication to read-back the Z-depth values would produce an intensive memory traffic with the subsequent energy cost, which makes this alternative less appealing.

Hence, we propose to integrate CD and image rendering within the GPU pipeline hardware. With minor adaptations, our technique reutilizes some intermediate results of the rendering pipeline to perform the CD task. These adaptations include allowing the software to pass collisionable object identifiers to the GPU, adding small and specific hardware (RBCD unit) to store depth-data and detect face intersections based on per-fragment location, and selectively deferring face culling. The GPU works as usual and performs the first two IBCD steps: projection and rasterization. Additionally, the special hardware we include in the GPU performs the last two IBCD steps: depth-sort lists of fragments and z-range overlap detection. Finally, the GPU only communicates the detected colliding pairs of points to the CPU. This way, the entire CD task gets seamless integrated into the graphics rendering pipeline of the GPU, hence we call it Render-Based Collision Detection. These adaptations are discussed below (implementation details are in Section 5.2):

1. Animation engines usually differentiate between collisionable and non-collisionable objects and, in order to be efficient, do not apply CD to the entire scene but just to the objects susceptible to collide. To identify collisionable geometry along the pipeline, each of the primitives that belong to collisionable objects must be associated to its object identifier. Besides, the application must send these identifiers to the GPU, embedded into the graphics commands. As will be shown later, no changes are needed to the OpenGL ES standard, but obviously existing software should be adapted to take advantage of the new GPU capability.

2. For each pixel, we need to store the z-depths of all fragments covering it, to make possible the z-overlap analysis. The conventional Z-buffer does not serve for this purpose because, since

its goal is to eliminate occluded surfaces, it just stores the z-depth of the front-most opaque fragment seen so far. Thus, we propose adding a new Z-depth Extended Buffer (ZEB), an array containing one entry per pixel, each having a list of z-depths instead of a single z-depth. For a Tile-Based Rendering (TBR) architecture such as our baseline (see Section 2.2.2) both the Z-buffer and the ZEB hold entries for just one tile of pixels, so they are implemented by means of fast, on-chip memory. Each ZEB entry is a fixed-size memory block containing a list of elements, each corresponding to a fragment and including not only its z-depth but also the object id and the front/back orientation tag. Moreover, to keep the lists always depth-sorted we need a new hardware block that stores every element from the Rasterizer into the corresponding ZEB list by following a simple insertion-sort algorithm.

3. Add a new specific hardware that analyzes the ZEB pixel by pixel (i.e., one list at a time), detects z-ranges overlaps, and sends the pairs of colliding points to the CPU, through system memory.

4. All the surfaces of collisionable objects must go through the Rasterizer stage, not only those that are visible, because all their fragments are needed to analyze possible z-range overlaps. The GPU includes a Face Culling (FC) stage where primitives that are identified as invisible are culled. In traditional GPUs, FC occurs early in the pipeline, before the Rasterization. Hence, culled primitives that belong either to front, back, or both faces, depending on the application, would never reach the Rasterizer nor the RBCD unit. We propose to defer the FC of collisionable objects after the Rasterization, when all fragment depths are already stored in the ZEB. We show later in Section 5.3.2 that this introduces a very small overhead, which is more than offset by the huge benefits of the proposed technique.

5.2 Microarchitecture

This Section introduces the extensions to the graphics pipeline of the baseline architecture that are required to support RBCD.

5.2.1 RBCD Overview

Figure 5.3 shows the GPU including the RBCD unit. This unit mostly works in parallel with other stages of the graphics pipeline, which introduces little peformance overhead in terms of GPU execution time. The rasterizer sends the fragments of collisionable objects to the RBCD unit ①. The RBCD unit applies an insertion-sort to store those fragments into the ZEB ②. Once all the collisionable fragments of a tile are in the ZEB, the RBCD unit performs the Z-overlap Test ③, and forwards the collision points to the CPU through the system memory ④.

5.2.2 Identification of Collisionable Objects

As stated above, identifiers are required by the RBCD hardware to determine whether two z-ranges belong to different collisionable objects. We need, therefore, to maintain the object identifier

Figure 5.3: GPU microarchitecture including an RBCD unit.

for the collisionable objects along the graphics pipeline up to the Rasterizer and the RBCD unit.

A simple way to do this is to include an object identifier in every draw command of collisionable objects. This could be done using an OpenGL extension based on the debug marker extension [191], implemented in OpenGL ES 1.1 and 2.0. This extension allows the programmer of an application to annotate the OpenGL command stream with a descriptive text marker. In our case, the marker would be the object ID and only the draw commands of collisionable objects would use it. This extension relies on the driver and the hardware to maintain the object notion through the rest of the graphics pipeline. Note that most of current 3D games already tag objects as collisionable or non-collisionable in order not to apply the CD algorithm to the whole universe of the scene but only to the objects that can collide. So the requirement here is to pass this information from the game and physics engine to the GPU.

Since the source code of the benchmarks studied is not available we could not automatically include object IDs for the collisionable objects of the analyzed scenes. Our strategy to assign object identifiers consisted on an off-line manual visual identification of the draw commands in every frame. For this purpose, we created an application in order to produce an independent image with the result of every draw command. Once identified, the GPU draw commands of every collisionable object were tagged with a unique object ID.

5.2.3 Deferred Face Culling

The FC stage, usually implemented with fixed-function hardware, is included in the Primitive Assembly stage of the Geometry Pipeline. FC may cull either the front, the back, or both faces of the geometry [104]. Since a fundamental requirement for our RBCD unit is to consider the complete geometric model of the objects, we propose to defer the culling of collisionable objects. During the FC stage, collisionable objects that are to be culled are just tagged-to-be-culled, so they can be later identified and handled appropriately after rasterization. The Tile Scheduler sends all the primitives of a given tile to the Rasterizer, which scan-converts them, creating fragments. The Rasterizer sends all collisionable fragments to the RBCD unit and it sends both collisionable and non-collisionable fragments to the Early Z-Test, except those tagged-to-be-culled, which are filtered out at this point, thus making the Deferred Face Culling take effect.

Obviously, this approach causes an increment in the number of primitives sent to the Rasterizer, and therefore in the number of fragments it produces. In addition, it causes an increment in the traffic (both writes and reads) from/to the Scene Buffer through the Tile Cache. However, this overhead does not affect further parts of the pipeline because the tagged-to-be-culled fragments never reach the Early Z-Test stage nor the Fragment Processors, which are identified as the most consuming part of the graphics hardware pipeline [176]. We analyze these effects in Section 5.3.2.

5.2.4 Insertion into the Z-depth Extended Buffer

RBCD requires that all fragments for a given pixel are already inserted into the ZEB buffer before the RBCD unit begins to perform the z-overlap analysis.

All the z-depths of collisionable fragments are sent to the RBCD unit, where they are first stored into the Z-depth Extended Buffer (ZEB). The ZEB is basically an on-chip buffer containing an array of lists, one list per pixel position. In our TBR architecture there are 16x16 pixels per tile, so the ZEB contains 256 lists. Each list contains M elements, each describing one point in the surface of a collisionable object. Each element in the list includes the z-depth, the object ID, and the front/back orientation tag. For instance, for M=8 the size of the ZEB would be 8 KB. The size M of the list is fixed for simplicity, which puts a limitation on the number of fragments per pixel that can be analyzed. Beyond that limit, an overflow occurs and some object overlaps could be lost. This is discussed in Section 5.3.3.

Since RBCD relies on z-ordered lists to detect object overlaps, the ZEB uses a simple sorted insertion policy entirely implemented in hardware, as shown in the block-diagram in Figure 5.4. The insertion of a new element E_{new} is a three-step process:

1. The list that corresponds to the currently processed pixel is read from the ZEB and stored in the List-Register. It contains up to M front-to-back ordered elements E_0 to E_{M-1} having depth values Z_0 to Z_{M-1}.

2. The depth Z_{new} of the element E_{new} is compared against all z-values of the list in the List-Register, which is done in parallel with an array of less-than comparators. Each comparator

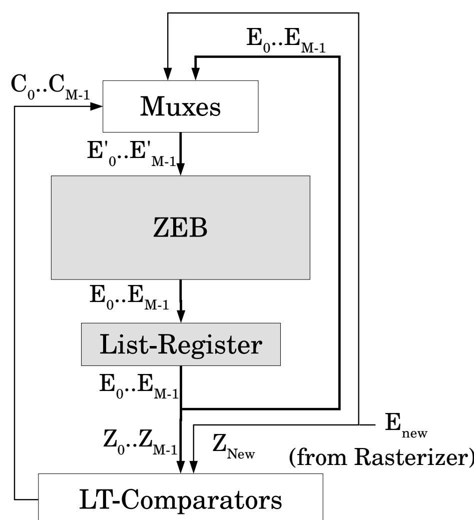


Figure 5.4: Sorted insertion hardware.

output C_i tells whether Z_{new} is less than Z_i, and bits C_0 to C_{M-1} are forwarded to an array of M Multiplexers that shifts some elements and places E_{new} in its corresponding order.

3. The ordered list, E'_0 to E'_{M-1}, is written back to the ZEB.

5.2.5 Z-Overlap Test

Once all the fragments of the tile are stored in the ZEB, the Z-Overlap Test sequentially reads the lists from the ZEB and for each list it analyzes possible overlaps of z-ranges between different objects. The z-depth along with the 2D coordinates of all the fragments form a 3D representation of the 3D scene, which makes the RBCD be projection-independent. The interference cases between two objects are illustrated in Figure 5.5. Each list illustrates one case, with points from A and B z-ordered front-to-back. The algorithm traverses each list front-to-back (left-to-right), and takes the corresponding actions. Colliding pairs are detected in cases 2 and 3.

The detection hardware is depicted in Figure 5.6. The algorithm begins reading one ZEB entry and storing it in the List-Register, then it traverses that z-ordered list front to back, analyzing each element in sequence by comparing it with the content of the FF-Stack. The FF-stack is a small table containing up to T entries, each having the object-id of a front-face fragment, and a matched bit M_i that indicates whether the element has already been paired with a back-face.

At the beginning, the FF-Stack is empty, then each element of the List-Register is read in turn and analyzed. If the current element E_{cur} belongs to a front-face, then Id_{cur} is pushed onto the FF-Stack and its M bit is initialized to 0. Otherwise, if E_{cur} belongs to a back-face, then the following two steps are done:

1. Id_{cur} is compared against the object-ids of all elements in the FF-Stack with M_i=0, in search of its corresponding front-face. The bottommost matching id, Id_m, of the FF-Stack delimits

Case	List	E_{cur}	Action	Stack	Matched	Notify
1.-	$[_A\ ]_A\ [_B\ ]_B$	$[_A$	push	$[_A$	0	
		$]_A$	match	$[_A$	1	
		$[_B$	push	$[_A, [_B$	1, 0	
		$]_B$	match	$[_A, [_B$	1 ,1	
2.-	$[_A\ [_B\ ]_A\ ]_B$	$[_A$	push	$[_A$	0	
		$[_B$	push	$[_A, [_B$	0, 0	
		$]_A$	match	$[_A, [_B$	1, 0	<A,B>
		$]_B$	match	$[_A, [_B$	1, 1	
3.-	$[_A\ [_B\ ]_B\ ]_A$	$[_A$	push	$[_A$	0	
		$[_B$	push	$[_A, [_B$	0, 0	
		$]_B$	match	$[_A, [_B$	0, 1	
		$]_A$	match	$[_A, [_B$	1, 1	<A,B>

Figure 5.5: Interference cases between front-faces ([) and back-faces (]) of two objects, A and B.

a depth interval, (Id_m, Id_{cur}). All the front-faces Id_i that stay above Id_m in the FF-Stack, regardless of the bit M_i, belong to the overlapping interval, so their output bits Hit_i are set to 1 to notify that there are collisions between the objects Id_i and the object Id_{cur}.

2. All the colliding pairs $<Id_i, Id_{cur}>$ and their coordinates are written to an output buffer which will be sent to the memory controller. The matched bit M_m of the front-face element Id_m is set to 1 before the algorithm continues traversing the list. Tagging the elements as previously matched instead of deleting them from the FF-stack not only simplifies the stack management, but also allows the detection of overlaps with following back-faces of the list.

Note that one ZEB cannot contain fragments of different tiles at the same time, so whenever a new tile is going to be sent to the Rasterizer, if the ZEB is still receiving fragments of the previous tile, or is being analyzed by the Z-overlap Test, then the Tile Scheduler must wait until the CD analysis completes, causing a pipeline stall that may hurt performance and therefore increase the static energy cost for the entire Raster Pipeline. Fortunately, the hardware needed to implement a ZEB is not expensive, so the RBCD unit is equipped with two ZEBs in order to store the information from the current tile being rasterized while the Z-overlap test is still being run for the previous one.

5.2.6 Animation Loop

The application stage of the conventional graphics pipeline, usually executed in the CPU, is responsible for receiving user inputs, detecting collisions, computing the corresponding reactions based on physical rules and updating the scene accordingly. This set of tasks constitute a time step.

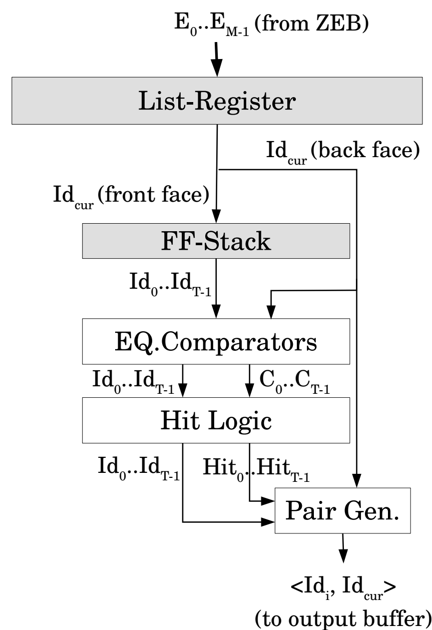


Figure 5.6: Z-overlap Test hardware.

After one or multiple time steps, the resulting scene is finally rendered, usually with the support of the GPU. After this, the whole process is repeated again, completing an animation or game loop [181]. Figure 5.7 shows, for two consecutive frames, an example of game loop without and with RBCD.

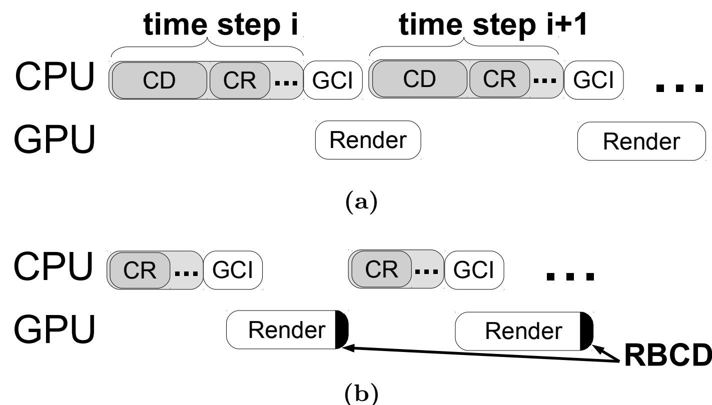


Figure 5.7: Example of game loop execution in the CPU/GPU system, (a)without RBCD, (b)with RBCD. CR and GCI stand for Collision Response and GPU Command Issue respectively.

Our proposal enables moving the CD task from the time step to the GPU rendering. Hence, per every rendered frame, RBCD detects collisions between all the collisionable objects sent to the GPU, visible or not. Should the application run additional time steps, it can be done by rasterizing (not fragment processing) extra commands just containing the collisionable objects to be tested, or by calling conventional software-based CD. Similarly, both approaches can be used to deal with

collisionable objects out of the view frustum if needed.

Executing multiple steps per frame can help improve realism of the animations, especially when rendering off-line or using high performance graphics. However, because of the power limitations of real-time rendering in mobile devices, one time step per frame is what actually occurs in most current real-time applications.

5.2.7 Power model of the RBCD unit

The RBCD unit has been modeled using McPAT's components, shown between parenthesis in the following list: the ZEBs (SRAM), LT Comparators (ALU); EQ Comparators (XOR); List-Register, FF Stack, list and stack pointers (registers); hit logic (priority encoder); and MUXes (MUX). The total area of the RBCD unit is less than 1% of the area of the GPU.

5.2.8 CPU Collision Detection Simulation

Since we do not have the source code of the benchmarks, in order to simulate the CD on the CPU for our benchmark set we extracted the geometry of all the collisionable objects for every GPU trace fed into the cycle-accurate simulator of Teapot. That is, for a given trace we obtain the 3D meshes of vertices of every collisionable object in the same world space coordinates as they have in the original benchmark, which is all the information that a CD algorithm needs in order to test overlaps.

In order to simulate the CPU CD of our benchmarks set, we employed Bullet [129], a 3D Real-Time Multiphysics Library. Bullet provides state-of-the-art collision detection and soft and rigid body dynamics. Bullet is a good choice since it is widely used in industry [95] (e.g., Grand Theft Auto V and Red Dead Redemption).

Using Bullet, we have created an application that loads the meshes of the collisionable objects and then performs the CD for every frame of the original benchmark. We implement two different CD versions, the first one just performs a broad phase, and the second one performs both broad and narrow phases. These two versions are simulated with Marss86, which generates activity factors that are fed into McPAT to obtain the energy cost of performing the CD in the CPU. The time and the energy of loading the 3D meshes are subtracted from the CPU results.

5.3 Experimental Results

In this section, we first show the benefits of RBCD in a CPU/GPU system. Then we show the small overheads introduced in the GPU. Finally, we perform a sensitivity analysis to the maximum length of the ZEB lists. Table 5.1 lists the parameters used in our experiments.

Table 5.1: CPU/GPU Simulation Parameters.

GPU Parameters	
Tech Specs	400 MHz, 1 V, 32 nm
Screen Resolution	800x480 (WVGA)
Tile Size	16x16
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
L2 Cache	64 bytes/line, 8-way associative, 128 KB, 8 banks, 2 cycles
Color Buffers (4x)	64 bytes/line, 1-way associative, 1 KB, 1 bank, 1 cycle
Z Buffers (4x)	64 bytes/line, 1-way associative, 1 KB, 1 bank, 1 cycle
Non-programmable stages	
Primitive assembly	1 triangle/cycle
Rasterizer	4 fragments/cycle
Early Z test	8 in-flight quad-fragments, 1 Z-buffer
Programmable stages	
Vertex Processor	1 vertex processor
Fragment Processor	4 fragment processors
Latency Main memory	50-100 cycles
Bandwidth	4 bytes/cycle (dual channel)
RBCD Unit	
ZEB buffers (2x)	32 bit/element, 8 element/entry, 256 entries, 8 KB
CPU Parameters	
Tech Specs	1500 MHz, 1 V, 32 nm
2 Cores	1 MB L2
Core Parameters	
CPU Architecture	Harvard
L1 Instruction Cache	32 KB/Core
L1 Data Cache	32 KB/Core

5.3.1 Performance and Energy Consumption Benefits

This section demonstrates that RBCD provides huge benefits in terms of both performance and energy at a very low cost in a low-power CPU/GPU system, using a set of commercial graphic applications (see Table 5.2). The speedup (5.1) is obtained comparing the time that the CPU employs performing the CD against the extra time that RBCD adds to the baseline GPU execution time. The energy consumption is computed using equation (5.2). On the CPU side, the energy

Table 5.2: Benchmarks.

Benchmark	Alias	Description
Captain America	Cap	beat'em up
Crazy Snowboard	Crazy	snowboard arcade
Sleepy Jack	Sleepy	action
Temple Run	Temple	adventure arcade

numbers include the energy consumption of the CPU and the main memory. On the GPU side, the energy numbers also include the consumption of the GPU and the main memory.

$$Speedup = \frac{t_{CPU_{CD}}}{t_{GPU_{RBCD}} - t_{GPU_{baseline}}} \tag{5.1}$$

$$Normalized\ Energy\ Consumption = \frac{E_{GPU_{RBCD}} - E_{GPU_{baseline}}}{E_{CPU_{CD}}} \tag{5.2}$$

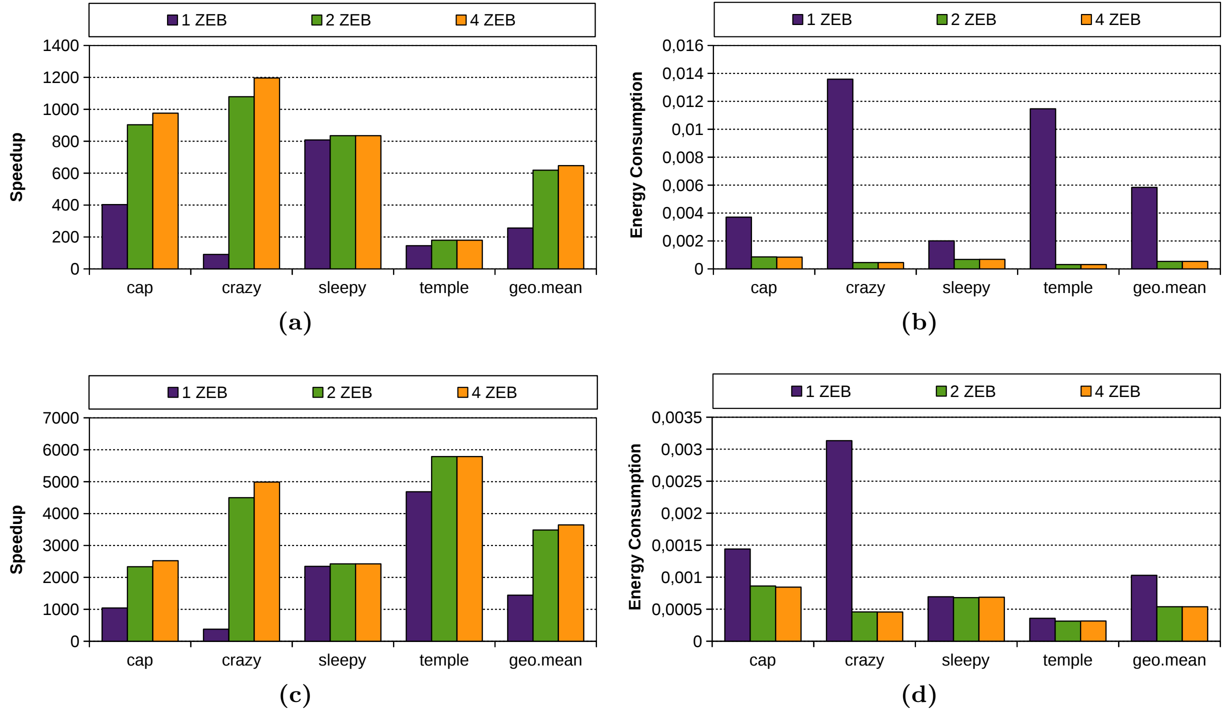


Figure 5.8: (a) RBCD speedup regarding Broad-CD, (b) Normalized energy consumption of RBCD regarding broad-CD, (c) RBCD speedup regarding GJK-CD, (d) Normalized energy consumption of RBCD regarding GJK-CD

Figures 5.8a and 5.8b show the speedup and the normalized energy consumption of RBCD with respect to a system that only performs a broad CD using AABBs in the CPU (Broad-CD). For these experiments, the RBCD unit is equipped with different number of ZEBs (1, 2 and 4). As it

can be observed, RBCD obtains a speedup of around 250x on average with just one ZEB buffer in the RBCD unit. As it is explained in the following section, including more ZEBs in the RBCD unit avoids some GPU stalls, which reduces the GPU time overhead caused by RBCD. With two ZEBs the speedup of RBCD w.r.t Broad-CD is incremented to around 620x, while with four ZEBs it is around 650x. Regarding energy, with one ZEB in the RBCD unit, RBCD consumes on average 0.6% of the energy consumed by Broad-CD in the CPU. Including a second ZEB further reduces the energy consumption because the reduced pipeline stalls save much more energy to the whole GPU than the tiny extra consumption of the additional ZEB. However, with 4 ZEBs the energy consumption is similar to the energy consumption with 2 ZEBs, because most of the stalls are already removed with 2 ZEBs, and the cost of including two more ZEBs is similar to the potential remaining savings.

RBCD is also compared against a scheme executing both the broad and narrow phases. The broad phase uses AABBs and the narrow phase is the Gilbert-Johnson-Keerthi algorithm (GJK [140, 194]) implemented in Bullet [129]. As shown in Figures 5.8c and 5.8d, the benefits in speedup and energy consumption are even higher, with speedups on average around 1400x, 3500x, and 3650x for 1, 2 and 4 ZEBs respectively. The energy consumption is around 0.1% on average when including 1 ZEB, and drops to 0.05% including 2 and 4 ZEBs respectively. Furthermore, the RBCD scheme provides a pixel-level granularity for CD, whereas the baseline broad-CD implements the most simple broad phase, an AABB overlap test. The benefits of RBCD are mainly due to:

1. Both the execution time and the energy of the CPU/GPU system are highly dominated by the CD part that executes in the CPU. RBCD removes all CD CPU activity while it requires minimal extra work in the GPU due to the reuse of data already computed during the image rendering process.

2. The power of the RBCD unit is very small compared with the total power of the GPU.

5.3.2 GPU Overheads

This section analyzes the overhead, in terms of execution time and energy consumption, of integrating RBCD into the rendering pipeline. Figures 5.9a and 5.9b show the execution time overhead (equation 5.3) and energy consumption (equation 5.4) of the proposed approach normalized to the baseline GPU.

$$Normalized\ Time = \frac{t_{GPU_{RBCD}}}{t_{GPU_{baseline}}} \tag{5.3}$$

$$Normalized\ Energy = \frac{E_{GPU_{RBCD}}}{E_{GPU_{baseline}}} \tag{5.4}$$

As shown in Figure 5.9a, the time overhead introduced by RBCD is on average less than 5.4% with 1 ZEB, it decreases to 3% with 2 ZEBs, and remains around 3% with 4 ZEBs. As stated before, since the ZEB can only hold data from just one tile at a time, having a single ZEB may force the Rasterizer to stall if it fills up the RBCD unit input queue with fragments from a new tile while the

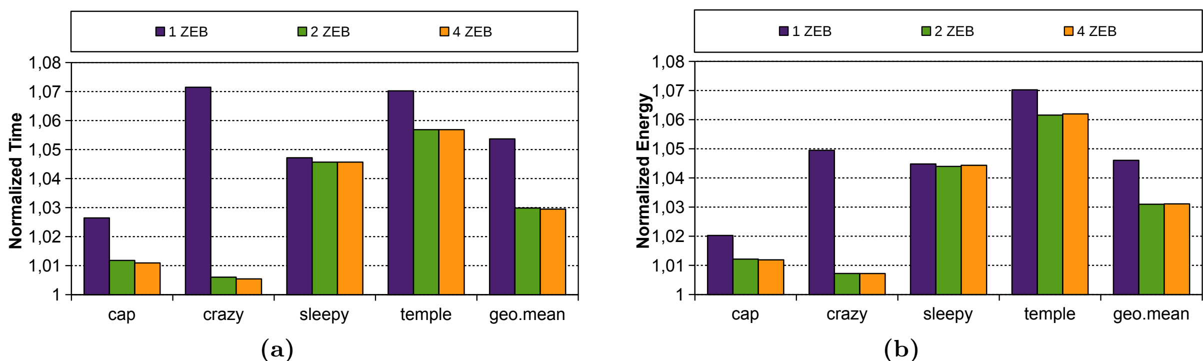


Figure 5.9: (a) Normalized rendering time of the GPU with RBCD w.r.t. the baseline GPU. (b) Normalized energy of the GPU and main memory with RBCD w.r.t. the baseline GPU.

Z-Overlap Test of the previous tile is not yet finished or even it has not started. Note, however, that the Rasterizer stalls may not have a direct impact on the total pipeline time if the queue that feeds the fragment processors contains enough work to keep them busy during the stall. On the contrary, if this queue becomes empty then the fragment processors are left idle, which produces a performance penalty. Therefore, having a second or more ZEB buffer(s) allows the Rasterizer to send collisionable fragments to another ZEB of the RBCD unit while the other ZEB is being used by the previous tile. The benchmark where this issue is most relevant is Crazy Snowboard, having an overhead around of 7% (the biggest) with one ZEB , and less than 1% (the smallest) with two ZEBs.

The energy overhead of both the GPU unit and the main memory is shown in Figure 5.9b. It is on average 5.1% with 1 ZEB and decreases to 3.5% with 2 ZEBs and 4 ZEBs. Figures 5.10a and 5.10b show in more detail the energy overhead induced by RBCD on the GPU and on main memory. Compared to the 2 ZEBs unit, with 4 ZEBs the energy consumption of the main memory slightly decreases in *cap* and *crazy* (see Figure 5.10b), because the execution time decreases too, i.e., the static energy consumption decreases. However, as can be seen in Figure 5.10a, with 4 ZEBs the energy overhead increases for all the benchmarks (even in *cap* and *crazy*), because the static energy cost of including 2 more ZEBs (4 in total) is greater than the reduction of static energy of the total GPU. An energy breakdown of the baseline is shown in Figure 5.11a.

In summary, we can conclude that two ZEBs are enough to avoid practically all stalls, because including 4 ZEBs or more (not shown in this graph) does not significantly improve time and slighly increases the energy consumption.

The second source of overhead comes from the extra tagged-to-be-culled primitives that are processed by the GPU pipeline, and which are necessary to perform CD in the RBCD unit. This extra work, which we examine in more detail in the next paragraphs, affects both the Geometry Pipeline and the Raster Pipeline but the main impact occurs in the latter one since its computing requirements are much higher, as shown in Figure 5.12. For these experiments we have considered two ZEBs in the RBCD unit.

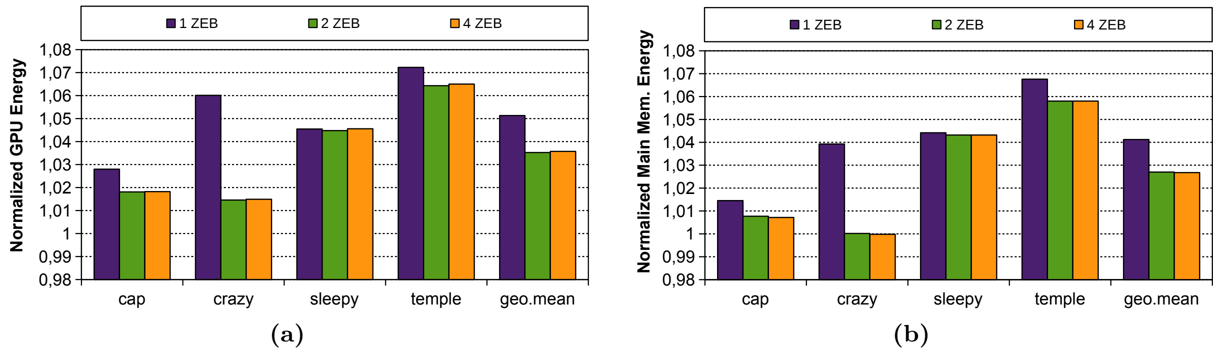


Figure 5.10: (a) Normalized energy consumption of the GPU with RBCD w.r.t the baseline GPU. (b) Normalized main memory energy of RBCD w.r.t. the main memory energy of the baseline GPU.

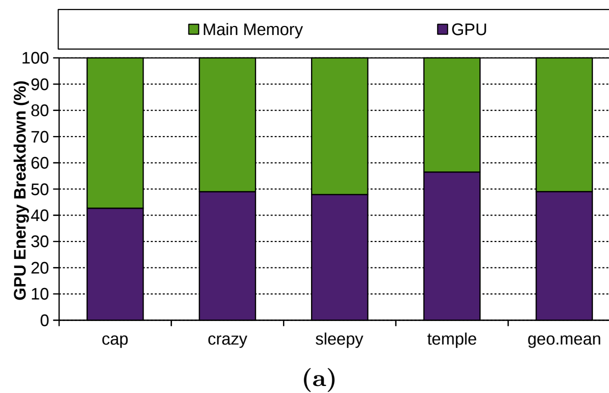


Figure 5.11: Energy GPU/Main memory breakdown.

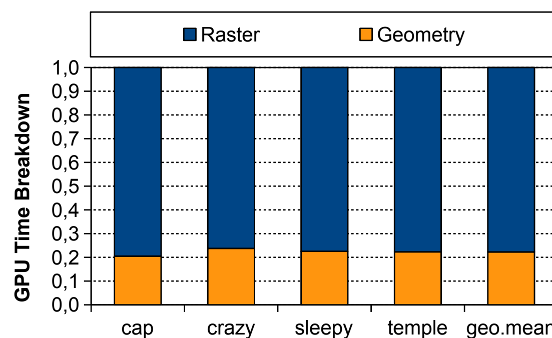


Figure 5.12: GPU time breakdown including time of Geometry and Raster pipelines.

Regarding the Geometry Pipeline, the Polygon List Builder has more work to do in order to process the extra collisionable tagged-to-be-culled primitives, which translates to 32% more stores to the Tile Cache and around 8.8% more write misses. However, the execution time of the Geometry Pipeline increases on average less than 1%.

Regarding the Raster Pipeline, Figure 5.13 shows some activity factors to illustrate the main sources of overhead, all normalized to the baseline GPU. The execution time of the Raster Pipeline increases by 3.7%. This is due to the fact that the Tile Fetcher reads 18.4% more primitives from the Tile Cache, which translates into 19.3% more loads and around 6.6% more read misses. The extra primitives cause the Rasterizer to produce 6.3% more fragments. However, the increase in the number of fragments is smaller than the increase of primitives because the average size of the tagged-to-be-culled primitives, which are part of high detail 3D models, is smaller than the average size of a primitive.

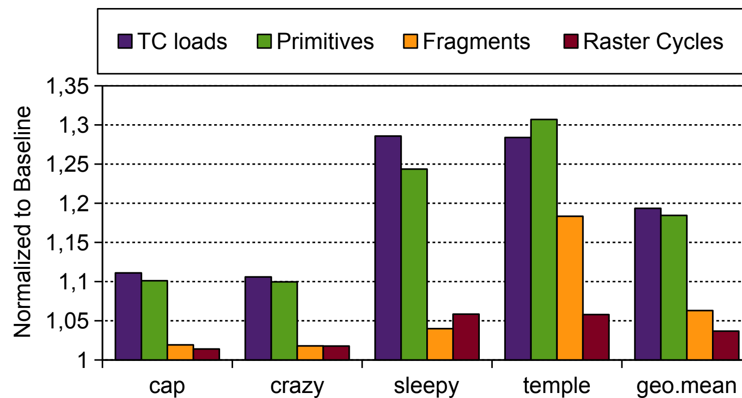


Figure 5.13: Tile Cache loads, primitives, fragments, and Raster Cycles with the GPU including RBCD normalized to the GPU baseline.

On the other hand, these extra tagged-to-be-culled fragments are not sent to the input queue of the Early-Z Test, which translates to an increment around 3.6% of the idle cycles of the fragment processors. The increment of idle cycles is smaller than the increment in the number of fragments, because the rasterization of the tagged-to-be-culled primitives partially overlaps with the execution of the fragment processors, thus partially hiding its latency. In other words, the execution time of the Raster Pipeline is increased if the fragment processors do not have enough work to do because the input queue that feeds them has been emptied while the Rasterizer is producing the tagged-to-be-culled fragments, which occurs in 3.6% more cycles. In summary, we can conclude that RBCD adds a low overhead to the GPU because:

1. RBCD reuses all data transmission and geometry processing of collisionable objects already done by the GPU to render the image of a given scene.

2. RBCD exploits the fact that the 84.4% of the primitives are already rasterized in the baseline, producing the 94% of the fragments needed by the RBCD unit.

3. The extra cycles that RBCD adds to the GPU pipeline are partially hidden by the fragment processing execution.

4. The static power of the RBCD unit is very small, being less than 1% of the total static power of the GPU.

Given that the time overhead in the GPU is greatly amortized by the time savings in the CPU we do not consider to include a dedicated Rasterizer for processing collisionable backfaces. However, additional resources could be devoted to the Rasterizer in order to smooth the time overhead in a context with much more rasterization overhead.

5.3.3 Sensitivity to ZEB List Length

Table 5.3: Percentage of fragment overflow for a ZEB with 4, 8 or 16 entries (each entry holds data for one fragment).

Benchmark	4	8	16
cap	1.57	0.01	0
crazy	1.20	0.03	0
sleepy	5.87	0.21	0
temple	16.61	0.96	0
average	3.68	0.08	0

The amount of collisionable geometry of a benchmark and its concentration in the scene may stress the RBCD unit and increase the overheads in the GPU. For our set of benchmarks, we found that one RBCD unit with one Insertion Sort unit, one Z-Overlap Test unit, and two ZEBs with lists of a maximum size of 8 are adequate. The size of a ZEB with 256 lists, 8 entries per list, and 32 bits per entry is 8 KB. As described above, we implemented the ZEB as an array of fixed-length lists for simplicity reasons. However, the downside is that overflows are possible if more than M fragments from collisionable objects are found in the same pixel, being M the length of the lists. Of course, having longer lists reduces the probability that overflows occur, but it comes at the cost of more area and energy.

Table 5.3 shows the percentage of times a list of the ZEB overflows for lists with 4, 8, and 16 entries. With four entries the overflow rate is below 2% for Captain America and Crazy Snowboard, but increases above 7% in Sleepy Jack and up to 16.6% in Temple Run. The reason is that the first two benchmarks have less collisionable objects, and they are more spread across the projection plane. In other words, they have less objects overlapping the same pixels than in the other two benchmarks. On the other hand, Table 5.3 shows that just eight entries per list are enough to keep the overflow rate below 1% in the worst case and 0.08% on average. Despite the overflows, we verified that all the collisions are still detected. This is not surprising because there are multiple pixels per object so there are also multiple opportunities to detect the collisions between objects. Finally, with 16 entries, overflows do not happen at all for our set of benchmarks.

Nevertheless, there may be cases where a benchmark stresses the ZEB to the point that the overflow rate is very high, decreasing the quality of the CD. A fallback procedure can be adopted in these cases by notifying the event to the CPU, which would then perform the CD in the conventional way. Another possibility is to design a ZEB with several spare entries that could be dynamically allocated as extra space to create longer lists for these cases. In any case, the percentage of static power consumed by the RBCD unit with two ZEBs relative to the total GPU static power is less than 1% with lists of 8 entries, and less than 5% with lists of 64 entries. This means that the ZEB

has a low impact on the total GPU static power and, if needed, the size of the lists could be higher in order to minimize the number of executions of the fallback procedure without causing a great impact in the total power of the GPU.

5.4 Conclusions

In recent years, mobile platforms and smartphones have become ubiquitous, as well as powerful computational engines. Among all the capabilities of these systems, battery life and graphics are probably the most appreciated ones by consumers. Current tablets and smartphones contain a GPU that is widely used by applications such as browsers, image processing, video viewers and games. Graphics animation applications, and 3D games in particular, are one of the most downloaded application types, and besides a good graphic quality, they usually require a physics kernel. Collision Detection is often the most compute-intensive part of these physics kernels.

In this chapter we have presented a hardware scheme, RBCD (Render Based Collision Detection) to perform low-energy, high-fidelity Collision Detection, which is based on the observation that most of the computation required by an Image Based CD algorithm is also performed when the image is rendered by the GPU. This technique introduces a small overhead in the GPU, both in time (below 3% on average) and energy (3.5% on average), but it frees the CPU of the detection of collisions, resulting on average in a 448x reduction (i.e., by 99.8%) of the total energy consumed by the CD on the CPU. Additionally, since this scheme detects collisions at a pixel level, it provides higher accuracy than conventional CD algorithms for mobile platforms, which usually apply simplifications to the objects in order to reduce the computation required. Furthermore, RBCD is almost three orders of magnitude faster (a 600x speedup) than traditional, less-accurate approaches that run on the CPU.

6

Conclusions

In this chapter we summarize the main conclusions of the works presented in this thesis, as well as point out open-research areas for future work.

6.1 Conclusions

Let us give a brief review of the graphics pipeline. Figure 6.1 shows a simplified graphics pipeline. The GPU receives vertices and processes them in the Geometry Pipeline, which generates triangles. These are then discretized by the Rasterizer, which generates fragments that correspond to pixel screen positions. Then, fragments are sent to the Fragment Processing stage, which performs the required texturing, lighting and other computations to determine their final color. Finally, the Depth test compares each fragment's depth against that already stored in the Depth Buffer to determine if the fragment is in front of all previous fragments at the same pixel position.

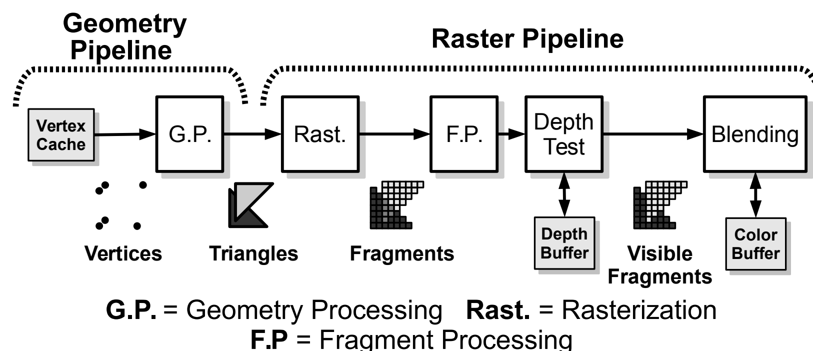


Figure 6.1: Simplified version of the Graphics Pipeline.

In first place, we evaluate the amount of overshading in mobile graphics workloads and the effectiveness of Early-depth test to reduce it. Early-depth reduces and important part of the overshading but there is significant room for improvement because, to be effective avoiding the rendering of hidden surfaces, it requires opaque geometry to be processed in a front-to-back order. Hence, when the GPU realizes that an object or part of it is not going to be visible, all activity required to compute its color has already been performed, with the consequent waste of time and energy, especially in the Fragment Processing stage, which as we already showed in Section 1.2.2 (in line with previous works [172, 176]) is the most power consuming stage of the graphics pipeline. Z-prepass addresses overshading by performing two separate rendering passes with the GPU. First Z-prepass renders the geometry without computing the color of the fragments, just using a null fragment shader in the processors, to determine the visibility of the scene. On a second pass with the real shaders the Early-depth test will perform optimal culling, so overshading will be minimum. Unfortunately, this approach doubles several stages of the graphics pipeline like vertex processing, rasterization and visibility determination, which may offset the benefits of the technique. It is only effective for workloads with enough complexity where the overhead of the first rendering pass is compensated by large fragment computation savings, which is not usually the case on mobile applications. Like Z-prepass, Deferred Rendering (DR) is a hardware technique that avoids overshading by performing a hidden surface removal (HSR) phase. In contrast to Z-prepass, DR does not perform the geometry processing twice. However, DR still has a non negligible cost: either it introduces a barrier in the graphics pipeline because the Fragment Processing stage is not started until HSR has completely finished for a tile, or extra hardware is required to perform the HSR of a tile in parallel with the rendering of another tile.

We present VRO, a novel technique that effectively reduces overshading. VRO reorders objects front-to-back entirely in hardware to maximize the culling effectiveness of the Early-depth test and minimize overshading, hence reducing execution time and energy consumption. VRO exploits the fact that the objects in graphics animated applications tend to keep their relative depth order across consecutive frames (temporal coherence) to provide the feeling of smooth transition. VRO keeps visibility information of a frame, and uses it to reorder the objects of the following frame. Since depth-order relationships among objects are already tested by the Depth Test, VRO incurs minimal energy overheads. It just requires adding a small hardware to capture the visibility information and use it later to guide the rendering of the following frame. The overhead of this technique 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. For a set of unmodified commercial applications, VRO outperforms state-of-the-art techniques in performance and energy consumption by reducing the overshading without the need of an expensive HSR stage at fragment granularity. VRO is especially efficient for geometry-complex applications, which are expected to be the most common applications in mobile devices as they already are in desktops. VRO achieves a speed-up about 27% and an energy reduction around 14.8% compared to an ARM Mali-400 MP4-like GPU. VRO outperforms DR because the Visibility Rendering Order is created out of the critical path while DR introduces significant overheads to perform the HSR stage. Furthermore, we have evaluated VRO over an IMR GPU architecture. Like VRO, IMR-VRO includes a small hardware unit that stores the order relations among the objects of a scene of the current frame in a buffer. This information is used in the next frame to create a Visibility Rendering Order that this time guides the Command Processor. IMR-VRO outperforms state-of-the-art techniques in performance and energy consumption by reducing the overshading without the need of an expensive extra rendering pass to determine

visibility. IMR-VRO achieves up to 1.32x speed-up (1.16x on average) and down to 0.79x energy consumption (0.85x on average) with respect to the baseline IMR GPU. IMR-VRO is much more efficient than Z-Prepass because most of the computations required to create the Visibility Rendering Order are reused from the normal rendering, while Z-Prepass requires an extra renderization pass that introduces significant overheads.

In second place, we introduce Collision Detection (CD), one of the main and most important algorithms used in physics kernels and that identifies the contact points between the objects of a scene to determine when they collide. Graphics animation applications such as 3D games represent a large percentage of downloaded applications for mobile devices and the trend is towards more complex and realistic scenes with accurate 3D physics simulations. However, real-time highly accurate CD is very expensive in terms of energy consumption and this parameter is of paramount importance for mobile devices since it has a direct effect on the autonomy of the system. We give a brief introduction to a group of CD algorithms known as Image-Based CD (IBCD). These algorithms rely on the rasterization of the surfaces of the scene objects and the detection of their intersections based on the pixel depths of the rasterized fragments. We present our proposal: Render Based Collision Detection (RBCD), which belongs to this kind of techniques, that have been proposed to exploit the computing power of graphics processors and their ability to rasterize polygons efficiently.

RBCD is a novel energy-efficient high-fidelity CD scheme that leverages some intermediate results of the rendering pipeline to perform CD. RBCD is based on the observation that most of the tasks required for IBCD (e.g., vertex processing, projection, rasterization and depth test) are also performed during image rendering. Hence, we propose to integrate CD and image rendering within the GPU pipeline hardware, so that redundant tasks are done just once. With minor hardware extensions and minor software adaptations our technique reutilizes some intermediate results of the rendering pipeline to perform the CD task. Some of these adaptations include allowing the software to pass collisionable object identifiers to the GPU, selectively deferring face culling, and adding small, specific hardware to detect face intersections based on per-fragment location and depth. We show the benefits of RBCD in a CPU/GPU system. Although this technique introduces a small overhead in the rendering of a frame in the GPU, both in time (below 3% on average) and energy (3.5% on average), RBCD frees the CPU of the detection of collisions. Comparing RBCD with a conventional CD completely executed in the CPU, we show that its execution time is reduced by almost three orders of magnitude (600x speedup) than traditional and less-accurate approaches that run on the CPU. Most of the CD task of our model comes for free by reusing the image rendering intermediate results. Although not necessarily, such a dramatic time improvement may result in better frames per second if physics simulation stays in the critical path. However, the most important advantage of our technique is the enormous energy savings that result from eliminating a long and costly CPU computation and converting it into a few simple operations executed by a specialized hardware within the GPU. Our results show that the energy consumed by CD is reduced on average by a factor of 448x (i.e., by 99.8%). These dramatic benefits are accompanied by a higher fidelity CD analysis (i.e., with finer granularity), which improves the quality and realism of the application.

6.2 Future Work

In this thesis we mainly focus on 3D graphics workloads which are the most compute and power demanding graphics applications. However, given the popularity of 2D graphics applications, an interesting extension to the study performed with respect to the RBCD scheme presented in this thesis could be to evaluate it with 2D graphics workloads, which would require minimum modifications. For example, given that Face Culling would not be present in 2D graphics workloads we would not need to detect Z-ranges overlaps anymore, thus making substantially simpler the object overlap detection. Take into account that the ZEB buffer is still needed to support different depth layers in the scene, but given that the notion of front and back faces would not be necessary, the number of objects that could be stored in every position of the ZEB buffer would be halved, thus reducing the possibility of ZEB overflows.

Likewise, the VRO scheme presented in this thesis could also be evaluated with 2D graphics workloads without significant modifications. On the other hand, our current evaluation of VRO only reduces a specific kind of overshading. The overshading can be differentiated in two types: intra-object and inter-object overshading. The former is produced by auto-occlusions of an object. The latter is the overshading caused by occlusions between different objects. Our current evaluation of VRO only labels geometry at object-level granularity, so it is only reducing inter-object overshading. Although there are techniques that are complementary to VRO and that effectively reduce the intra-object overshading, we believe that VRO has still room for improving performance and energy savings not only by combining it with one of those techniques but also making use of a more sophisticated labeling of geometry. A clustering method would be applied over the static 3D models of the application. Then, every object would be labeled with a primary object-ID plus a secondary cluster-ID. Such labeling process would only be performed once. Then, minor modifications to take into account this two level labels would allow to not only reorder the geometry at inter-object level but also to reorder geometry at intra-object (cluster) level and, hence, potentially reduce intra-object overshading.

Despite Khronos releasing OpenGL ES 3.0/3.1/3.2 specifications in December 2013, March 2014 and August 2015 respectively [97, 98, 99], during most of the time of this thesis there were not available representative Android games using OpenGL ES 3.XX API. Furthermore, by the time of doing the research studies included in this dissertation Gallium (*softpipe*, *llvmpipe*, *swr*) did not include support for a huge number of features included in OpenGL ES 3.XX. We hope that forthcoming developments on Gallium support for OpenGL ES 3.XX will allow us to include support for them in Teapot in the future. Furthermore, with the inclusion of support for OpenGL ES 3.2 in Teapot we could be able to study tessellation, which increases the detail of the scene at geometry level. Tessellation is a stage of the graphics pipeline that reads primitives and creates new ones by subdividing the input primitive into new ones that are passed to following stage of the pipeline, the Geometry Shading stage. This subdivision of primitives may greatly increment the number of primitives to be processed in following stages of the pipeline. Given that this subdivision is performed before the binning stage of TBR GPUs, the cost of handling an overwhelming increment in the number of primitives to be stored in the Parameter Buffer may offset the benefits of TBR in terms of main memory bandwidth savings if the binning does not include significant changes. It would be rather interesting to study how tessellation impacts TBR GPUs, as well as propose new tessellation schemes specifically designed for such GPUs.

Frame-to-frame coherence denotes the fact that successive frames are likely to be very similar if the difference in time is small. In other fields, such as video encoding, frame coherence allows an efficient calculation and storage of video sequences [197]. The main goal of exploiting frame coherence is to expose redundant computations across different frames and reuse them, thus avoiding unnecessary re-computations. Besides, by making use of spatial and temporal coherence, one can use knowledge about previous frames to influence scheduling in order to improve energy and performance like we do with VRO. The trend is towards a higher frame rate which just would increase the frame coherence. In such context it is expected to have a growing number of regions of screen with the same colors in one frame and the next. We have studied if in a TBR GPU there are unmodified tiles relative to the corresponding tile of the previous frame. If so, we could reuse the previous rendered colors for that tile. We have performed a short study of potential for our set of commercial benchmarks and the results show that a significant amount of the tiles do not change from one frame to the next. Thus, all the fragments of these tiles that are sent to the Fragment Processors could be culled away and the frame still would be complete, so the potential of this technique is great both in terms of performance and energy. The idea is to avoid the rendering of equal tiles in a TBR GPU. The technique would generate a signature utilizing information of the Geometry Pipeline and the inputs of the Fragment Processing stage. Later, in the first stage of the Raster Pipeline, the signature of the current tile being rendered would be compared with the signature of the tile in the previous frame. If both signatures are the same the rendering is not performed, thus avoiding the execution of the costly Raster Pipeline for this tile.

One of the main concerns when rendering a frame is the rendering resolution. If the resolution is low the image quality may be compromised, especially in the high-frequency areas of the image (object borders, sudden changes in color or lighting) where image artifacts are more likely to occur. In order to avoid some artifacts there have been proposed techniques like super-sampling (SSAA), which consists of rendering the whole frame at a higher resolution and later perform a down-sampling. Despite this technique may greatly improve the quality of the image, it increments the amount of fragment shading and wastes resources when rendering low-frequency areas of the frame. Multi Sample Anti Aliasing (MSAA) is a technique that improves the quality image reducing the high cost of SSAA. MSAA samples the color of a pixel at the rendering resolution, but the resolution of the depth and the color buffers is incremented, so they can store several samples per pixel. MSAA reduces the amount of fragment shading with respect to SSAA but it stills increments the rendering costs for low-frequency areas of the image. In order to avoid wasting resources it would be interesting to detect low- and high-frequency areas of a frame, and use that information so that the rendering resolution of the following frame is adjusted (incremented or decremented) conveniently depending on the image frequency of the previous frame, so that the final image would potentially have similar image quality but at lower cost.

Appendices

Visibility Rendering Order on IMR GPUs

In the chapter 4 we introduce VRO, a technique that reorders the geometry of a frame at draw-command/object granularity that effectively reduces the work executed in the processors of a GPU. We evaluate VRO on top of a TBR architecture. In spite of that, our proposal is orthogonal to the type of rendering architecture and, therefore, it can also be implemented on top of an Immediate-Mode Rendering GPU (IMR GPU). In this chapter we evaluate VRO on top of an IMR GPU and we illustrate the usefulness of VRO using various unmodified commercial 3D applications for which VRO achieves up to 1.32x speed-up (1.16x on average) and down to 0.79x energy consumption (0.85x on average) over the baseline IMR GPU. Furthermore, we also compare VRO against Z pre-pass, a popular software technique to reduce overshading.

A.1 Immediate Visibility Rendering Order

In TBR GPUs the rendering of a frame is decoupled in two main pipelines: geometry and rasterization. The geometry pipeline performs all the geometry-related operations and sorts the geometry of the whole frame into fixed size tiles of screen pixels. Once the geometry pipeline has processed all the frame, the raster pipeline performs the rasterization and the fragment-related operations tile by tile, which allows to include local on-chip memories of the tile size. Hence, reads and writes to Color and Depth/Stencil buffers caused by overdraw are performed in these local on-chip memories. When the rendering of a tile finishes the local buffers are flushed to main memory. On the contrary, with IMR GPUs the corresponding primitives of a draw command are processed through the entire graphics pipeline stages as soon as they are generated. Given that with IMR the fragments of a draw command are not sorted in tiles they can be located at any pixel of the screen. In such a context the accesses to Depth/Stencil and Color buffers is done through caches, which

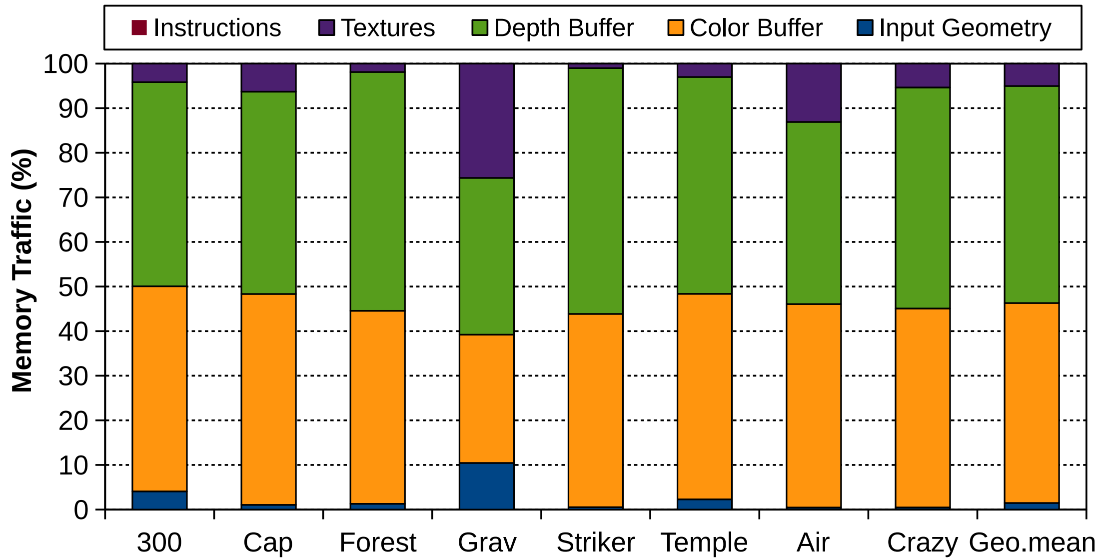


Figure A.1: Memory bandwidth usage on a mobile GPU for a set of commercial Android games. On average 98.5% of the bandwidth to main memory is caused by operations performed after rasterization.

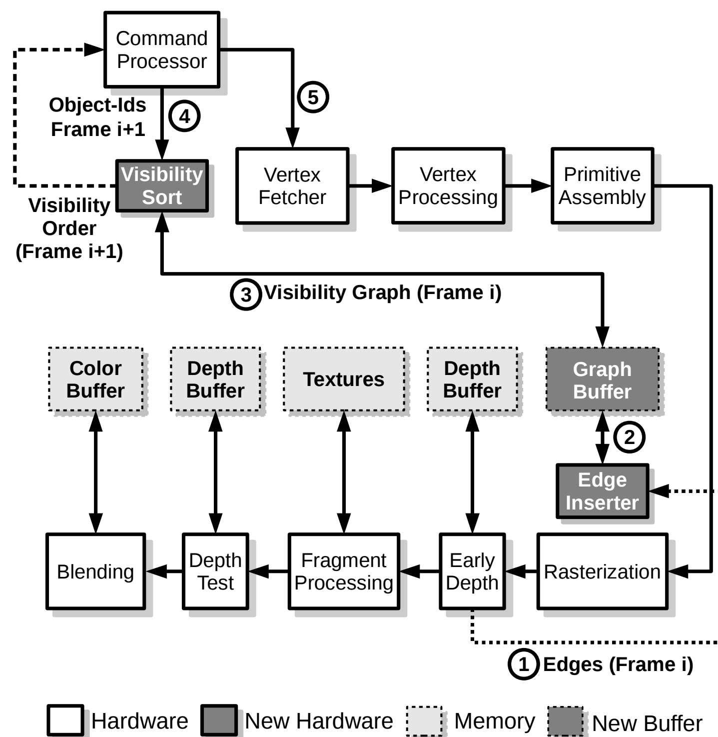


Figure A.2: Graphics pipeline including VRO.

increases the off-chip memory bandwidth and wastes energy. [1].

Figure A.1 shows, for a set of commercial Android games, the memory bandwidth usage in a mobile GPU similar to the Ultra Low Power (ULP) GPU included in NVIDIA Tegra SoC [70]. As can be seen, most of the traffic is caused by accesses to Color Buffer, Depth Buffer and Textures. All those accesses have in common that are performed by stages of the Graphics Pipeline that work with fragments. This is not surprising, because the number of elements (primitives) processed in the Geometry Processing stage is around 91 times smaller than the number of elements (fragments) processed in following stages of the Graphics Pipeline. For example, the instructions executed in the FPs represent around 94% of the total number of instructions executed in the GPU. Note that the Fragment Processing stage is the most power consuming stage of the graphics pipeline [176]. As it has been previously appointed in the introduction of Chapter 4, despite different stages of the Graphics Pipeline cull hidden surfaces, there is still room for improvement.

IMR-VRO can be divided in two conceptual stages that operate on consecutive frames: creation of the Visibility Graph and creation of the Visibility Rendering Order. As Figure A.2 shows, during the rendering of *frame i*, the Early-depth Test is used for visibility determination purposes as usual. Note that the comparisons performed produce ordered pairs of fragments (edges) covering the same pixel position. These edges, as precedence relationships that indicate which objects are in front of others, are sent to the Edge Inserter ①. The Edge Inserter uses the edges ② to build a Directed Graph (Visibility Graph), which represents the depth hierarchy of the objects in *frame i*. Once the depth of the objects in *frame i* has been completely tested, the Visibility Sort Unit traverses the graph ③ and creates a preliminary order which, along with the list of objects in *frame i+1* ④, is employed by the Command Processor to guide the rendering of the *frame i+1* ⑤.

A.1.1 Visibility Rendering Order Adjustments

The Command Processor receives the Visibility Rendering Order computed for the previous frame, which is sent by the Visibility Sort unit. At this point the Visibility Rendering Order only contains the object-ids of objects in the previous frame, and they may differ slightly from the ids of objects in the current frame. Note that the GPU driver usually buffers at least one frame before flushing to GPU [130, 92, 86, 90]. When all the draw commands of the new frame have been issued by the application and buffered by the GPU driver, the list of object-ids in the new frame is known and it is sent to the Command Processor. Therefore, like we said in Chapter 4, between the previous frame and the current one there may be two kind of disparities. On the one hand, objects present in the Visibility Graph but not present in the current frame. On the other hand, objects not present in the Visibility Graph but present in the new frame. The former ones are not scheduled by the Command Processor because they do not appear in the current frame. The latter ones must be scheduled by the Command Processor because they do appear in the current frame to be rendered. Since we do not have order information about those objects, we decide to schedule them after the objects in the graph.

Note that objects with Depth test disabled or with blending enabled cannot be put at the end of the order list because it could produce erroneous images, so these objects are scheduled in the

[1] Prior works point that accesses to main memory represent a large fraction of GPU energy consumption [102, 153]

same relative order as they appear in the program rendering order. VRO respects the OpenGL standard in the sense that the result is the same as if objects were processed in program rendering order. Fortunately, objects with depth disabled or blending enabled are commonly used by the GUI of applications and tend to be the last objects to appear in the program rendering order, so in practice they introduce minor constraints to the Visibility Rendering Order.

A.1.2 Visibility Rendering Order IMR GPU

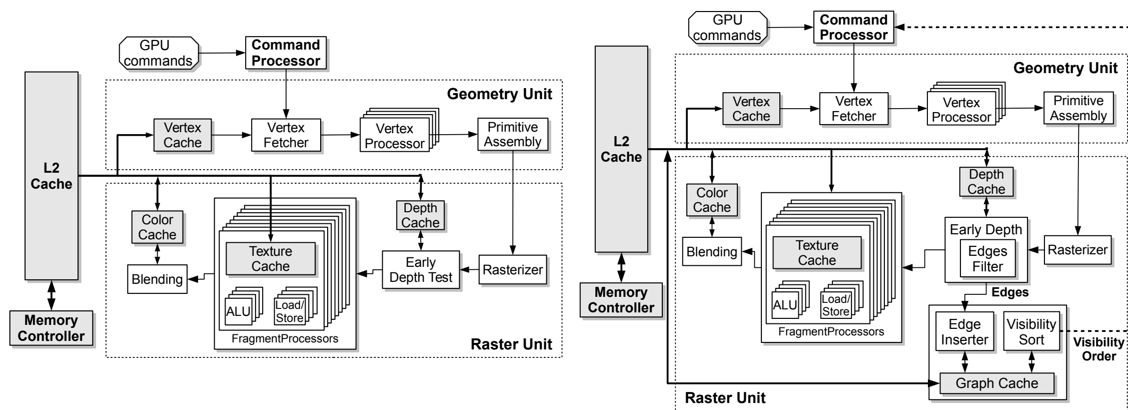


Figure A.3: Baseline IMR GPU architecture (left) and IMR GPU architecture including VRO (right).

As Figure A.3 shows, our technique is fully integrated in the graphics pipeline and includes several new pieces of hardware (see Section 4.3.2): the Edge Insertion unit, the Edge Filter, the Graph Cache and the Visibility Sort unit. Despite the following paragraphs of this section are similar to the ones employed to introduce VRO on a TBR architecture we believe that for the sake of clarity is better to include them.

On the one hand, the Edge Insertion for an edge is done after the comparison in the Early-depth Test of the two fragments, which determines the order relationship among them. This is done in parallel with the Early-depth of other fragments.

The Early-depth unit sends the edges to the Edge Inserter unit, which is responsible for storing them in the Graph Buffer, a buffer in main memory accessed through the Graph Cache, which contains the Visibility Graph of the frame being rendered. The edge insertions take place at fragment granularity using the results of Early-Depth comparisons. However, since graph edges represent object pairs there is a large amount of tests that actually produce the same edges. The Edge Filter is a small and fast associative on-chip structure that caches the most recently inserted edges and filters out redundant insertions to the Graph Buffer, thus avoiding unnecessary Graph Cache accesses. Thanks to this structure, the Edge Insertion unit accesses the Graph Cache much less than once every thousand fragments on average.

On the other hand, the Visibility Sort unit sorts the Visibility Graph and creates a preliminary ordered list of nodes, which is sent to the Command Processor. After the adequate adjustments to

satisfy the restrictions presented in subsection A.1.1, the Command Processor produces the final Visibility Rendering Order. Although other schemes could be adopted we perform these operations at the beginning of the rendering of the next frame. We have measured that on average the total time required for this process represents around 0.06% of the execution time of the graphics pipeline, so minimal overheads are introduced.

The Graphics Pipeline, instead of reading the draw commands in program rendering order, reads the primitives in Visibility Rendering Order, which increases the culling effectiveness of the Early-depth test reducing the amount of fragments that are processed in following stages of the Graphics Pipeline.

The IMR-VRO unit has been modeled using McPAT's components, shown between parenthesis in the following list: Graph Cache (Cache); EQ Comparators (XOR); Muxes (MUX); Min-Comparator (ALU); Adders (ALU); Subtractors (ALU); and registers. The area overhead of VRO is less than 1% (w.r.t. baseline).

A.2 Experimental Framework

We model not only the baseline GPU architecture, which closely resembles that of the ULP GeForce Tegra 3 architecture, but we also model IMR-VRO on top of the baseline GPU. In our experiments, we employ a set of benchmarks that is composed of eight different Android commercial 3D applications (see Table A.1).

Table A.1: Benchmarks.

Benchmark	Alias	Description
300	300	hack & slash
Air Attack	Air	flight arcade
Captain America	Cap	beat'em up
Crazy Snowboard	Crazy	snowboard arcade
Forest 2	Forest	horror
Gravity	Grav	action
Striker	Striker	first person shooter
Temple Run	Temple	adventure arcade

A.3 IMR-VRO Results

Figure A.4 shows the normalized speed-up achieved by our technique with respect to the baseline IMR GPU. As it is shown, IMR-VRO achieves up to 1.32x speed-up (*Striker*), and 1.17x% on average, being the lowest speed-up 1.015x (*300*). Figure A.5), shows the normalized energy consumption of IMR-VRO with respect to the baseline. As it can be observed, in the best case (*Striker*) IMR-VRO consumes up to 0.79x and 0.85x on average, being 0.96x in the worst case (*300*).

Table A.2: GPU Simulation Parameters.

Baseline GPU Parameters	
Tech Specs	400 MHz, 1 V, 32 nm
Screen Resolution	1200x768
Queues	
Vertex (2x)	16 entries, 136 bytes/entry
Triangle	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 (8x)	64 bytes/line, 2-way associative, 8 KB, 1 bank, 1 cycle
Color Cache (8x)	64 bytes/line, 2-way associative, 8 KB, 1 bank, 1 cycle
Depth Cache	64 bytes/line, 2-way associative, 4 KB, 1 bank, 1 cycle
L2 Cache	64 bytes/line, 8-way associative, 256 KB, 8 banks, 2 cycles
Non-programmable stages	
Primitive assembly	1 triangle/cycle
Rasterizer	4 attributes/cycle
Early Z test	32 in-flight quad-fragments, 1 Depth Buffer
Programmable stages	
Vertex Processor	4 vertex processors
Fragment Processor	8 fragment processors
Latency Main memory	50-100 cycles
Bandwidth	4 bytes/cycle (dual channel)
Extra Hardware IMR-VRO GPU	
Edges Filter	32 elements, LRU, 1 cycle
Graph Cache	64 bytes/line, 4-way associative, 4 KB, 1 bank, 1 cycle
Edge Insertion	1 Edge Inserter unit
Graph Sort	1 Visibility Sort unit
Edges Queue	64 entries, 4 bytes/entry
Order Queue	64 entries, 2 bytes/entry

The speed-up and the reduction in the energy consumption are directly caused by the fact that IMR-VRO processes the GPU commands in Visibility Rendering Order, which increases the culling rate of the Early-Depth stage avoiding further processing of non-visible fragments. Figure A.6 shows the energy breakdown of IMR-VRO with respect to the baseline GPU. As can be seen, most of the energy consumption is caused by the main memory and the Fragment Processors. Figure A.7

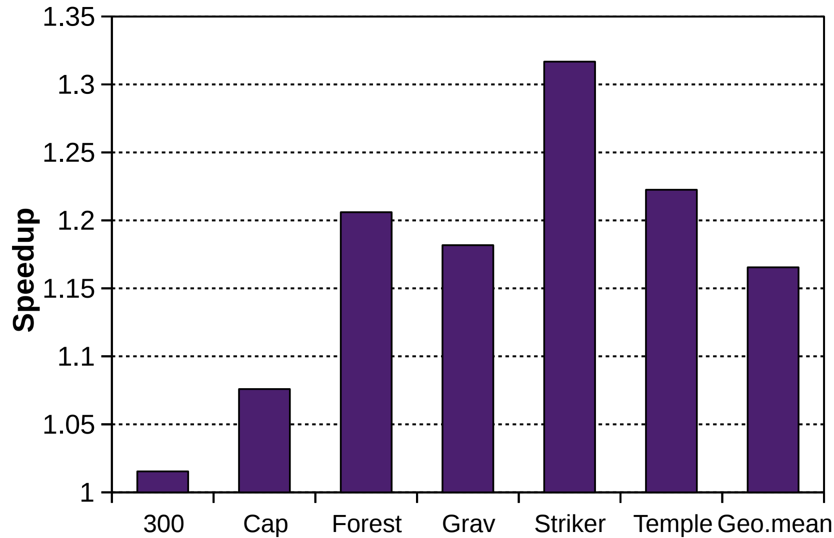


Figure A.4: Speed-up of IMR-VRO normalized to the baseline IMR GPU.

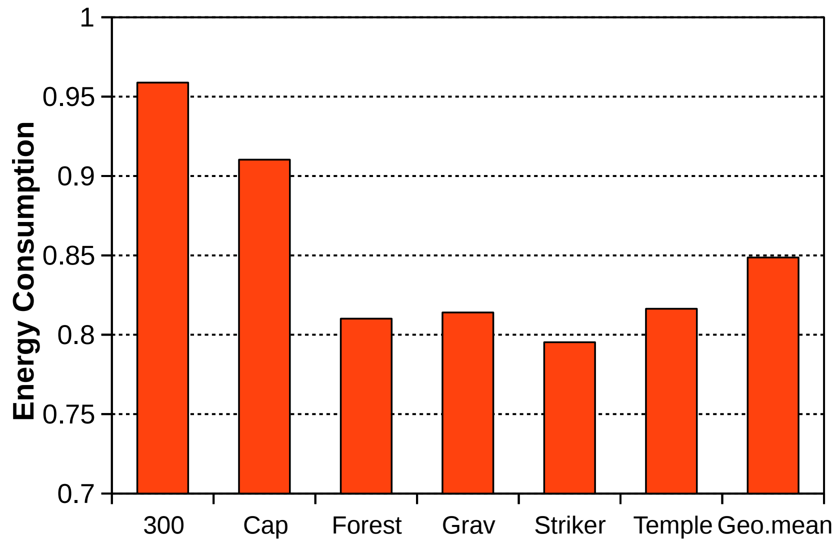


Figure A.5: Energy consumption of IMR-VRO normalized to the baseline IMR GPU.

shows both the overshading and the number of instructions executed in the Fragment Processors of IMR-VRO normalized to those of the baseline. As it can be observed the first bar closely correlates with the speed-up and the energy consumption shown in Figures A.4 and A.5. On average, the number of fragments that pass the Early-Depth test decrease to 0.77x, being 0.943x in the worst case (*300*) and 0.64x in the best case (*Striker*). Likewise, the number of instructions executed in the Fragment Processors decreases to 0.71x on average, being 0.938x in the worst case (*300*) and 0.54x in the best case (*Forest*), which is translated into significant energy savings. Note that the first and the second bar of Figure A.7 show different reductions because the objects in a scene may execute a different Shader Program (with different number of instructions).

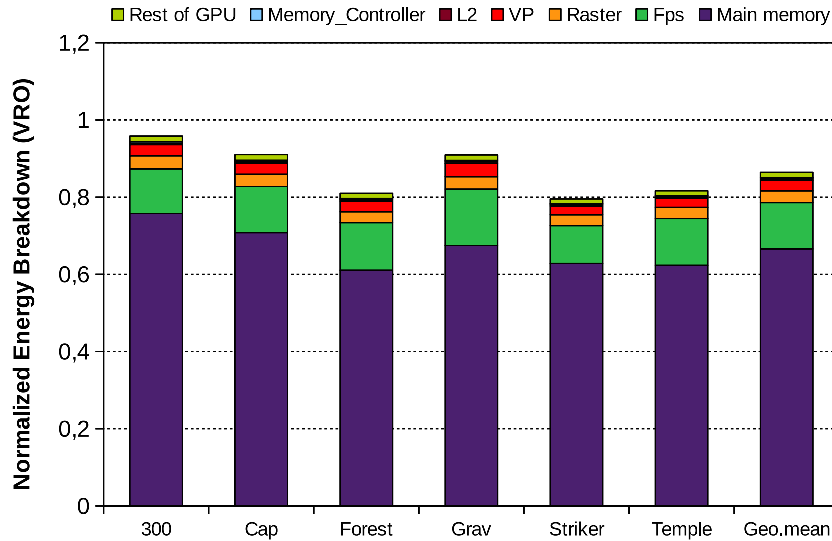


Figure A.6: Energy breakdown of IMR-VRO normalized to the baseline IMR GPU.

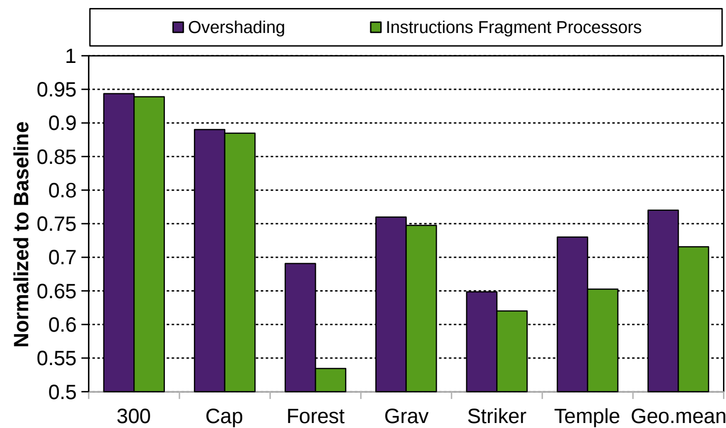


Figure A.7: Overshading and Instructions Executed in the Fragment Processors with IMR-VRO normalized to those of the baseline IMR GPU.

Figure A.8 shows the ratio between the time savings (produced by the overshading reduction) and the time overhead (produced by the computation of the Visibility Rendering Order) with VRO. As can been seen, the time and energy savings of IMR-VRO far exceed the cost of producing the Visibility Graph and the Visibility Rendering Order.

Take into account that IMR-VRO not only reduces the execution time but also reduces the energy consumption of the system. Energy Delay Product [159] is a metric that evaluates the energy efficiency of a system. Figure A.9 shows the Energy Delay Product (EDP) of IMR-VRO normalized to the EDP of the baseline GPU.

As it has been previously appointed, a huge part of the traffic with main memory is produced

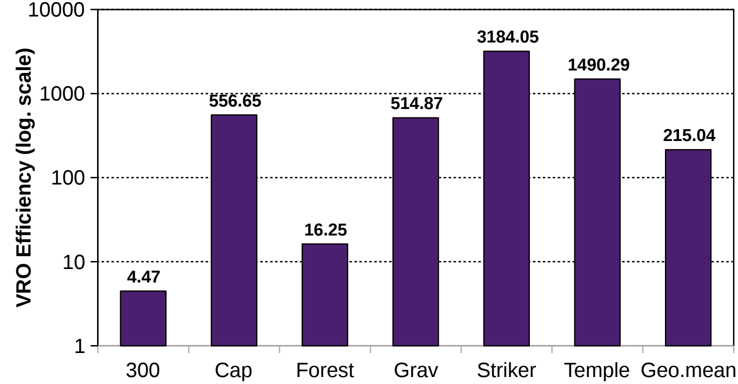


Figure A.8: Ratio between the time savings and the time overhead of VRO (higher is better).

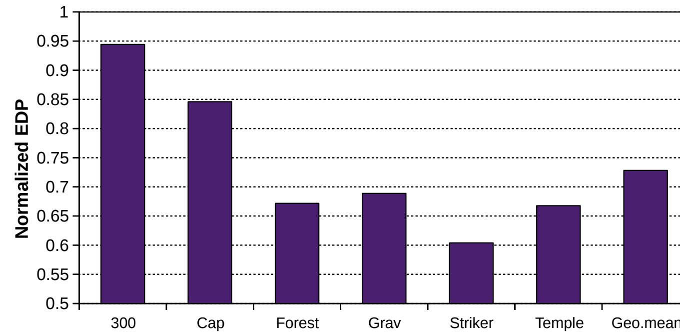


Figure A.9: Energy Delay Product of IMR-VRO normalized to the baseline IMR GPU (lower than 1 is better).

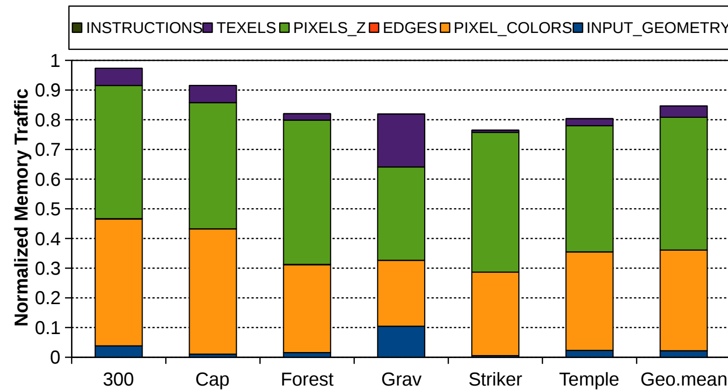


Figure A.10: Memory bandwidth usage of IMR-VRO normalized to baseline IMR GPU.

by operations related to fragments (98.5% on average). Recall that in our experiments the Graphics Pipeline processes around 91x more fragments than primitives. Therefore, the decrease in the

number of fragments that pass the Early-Depth test also produces a significant reduction in the main memory traffic up to 23.5% and 15% on average (see Figure A.10). This reduction is due to the fact that IMR-VRO reduces overshading and, hence, it also reduces main memory accesses performed in the Fragment Processing stage of the graphics pipeline: reads/writes to Color Buffer, writes to Depth Buffer and reads of textures among others. Figure A.11 shows stats of the caches related to those buffers as well as the L2 cache. As can be seen the number of misses of the Color, Depth and Texture caches is reduced in all the cases. The number of writebacks of the Color and Depth caches is reduced too. These decrements cause a reduction in the number of misses and writebacks of the L2 cache, which ultimately reduces the traffic with main memory.

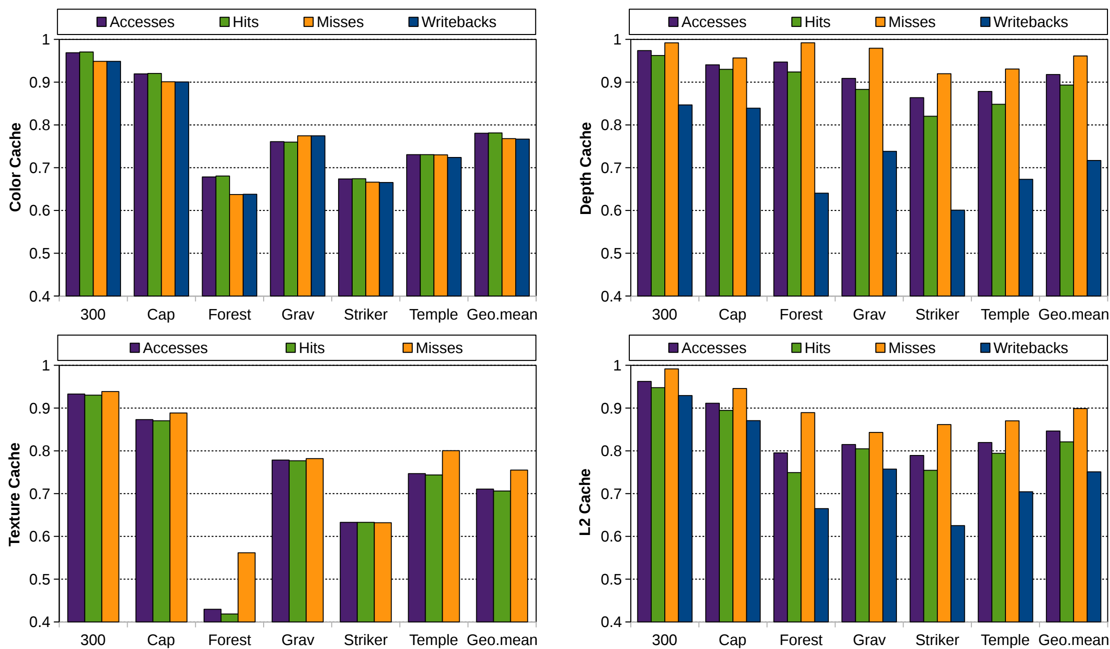


Figure A.11: Activity factors with IMR VRO normalized with those of the baseline IMR GPU for Color cache (top left), Depth cache (top right), Texture Caches (bottom left) and L2 cache (bottom right).

A.3.1 Software Z-Prepass

Z pre-pass (a.k.a Depth pre-pass) is a common software approach aimed to reduce overshading that has gained pace in the last years and it can be considered the standard on IMR and TBR GPUs to reduce overshading in complex games with multiple dynamic objects and costly Fragment Shaders. Some vendors like ARM recommend to use it when setting the rendering order of the objects is not possible because the complexity of the scene:

"*For example, if you have a set of objects with computationally expensive shaders and the camera can rotate around them freely, some objects that were at the back can move to the front. In this case, if there is a static rendering order set for these objects, some might be drawn last, even if they are occluded. This can also happen if an object can cover parts of itself.*" [4]

Z pre-pass exploits the Early-depth test by means of two rendering passes. The first pass performs pipeline stages up to the Early-deph test, which stores the depths of the visible fragments in the Depth Buffer. In the second pass, the full pipeline is executed and only the visible fragments pass the Early-depth, so only visible fragments are executed in the Fragment Processors. This technique doubles the cost of some stages of the pipeline (see Figure A.12), which is unacceptable in many scenarios. Taking into account that even if the benefits of Z pre-pass are greater than its cost, the extra depth pass represents a large portion of the total rendering time. For example, Z pre-pass can be found on The Blacksmith [6], which is a cutting edge real-time demo of the last version of Unity [21] made to show the most advanced graphics features that the game engine offers. We have studied three different frames of Blacksmith, where Z pre-pass represents 41.4%, 29.3% and 26.1% of the total rendering time respectively.[2]

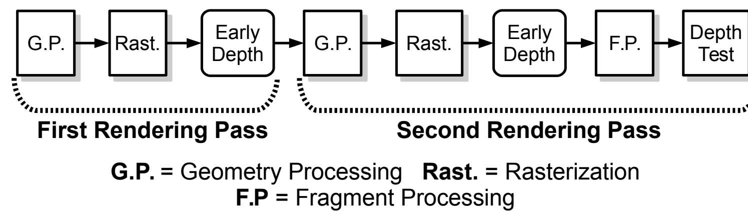


Figure A.12: Simplified version of the Graphics Pipeline executing Z-Prepass.

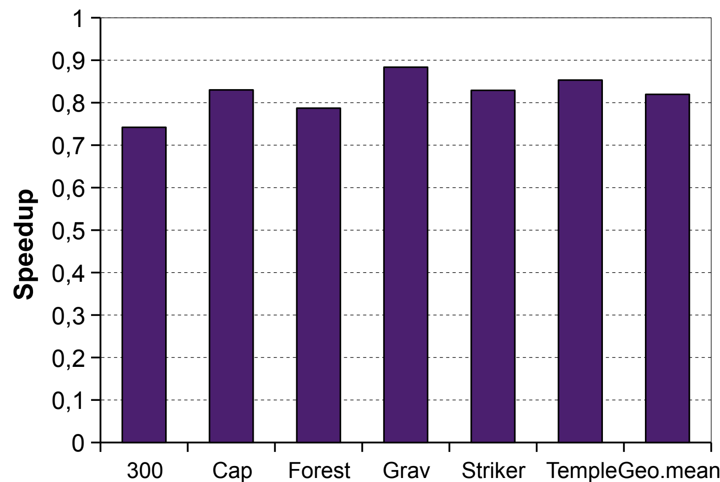


Figure A.13: Speed-up of Z pre-pass normalized to the baseline IMR GPU.

Given that Z pre-pass has the same target as VRO we have conducted an experiment to measure its effectiveness in common mobile graphics workloads. Figures A.13 and A.14 show the speedup and energy consumption of Z pre-pass. As can be seen, Z pre-pass introduces a slowdown in all the cases (0.82x on average). Regarding energy, in *Forest 2*, *Gravity* and *Temple Run* Z pre-pass reduces the energy consumption in the GPU less than 5%, but it increases the traffic with main memory by almost 20% (see Figure A.16), which along with the extra exeuction time, also increases the total energy consumption in all the cases (1.28x on average). Likewise, in the other benchmarks the increase in the main memory traffic and the execution time greatly penalize the energy consumption.

[2] Real hardware measures reported by Renderdoc [20] using an NVIDIA GTX 950 GPU.

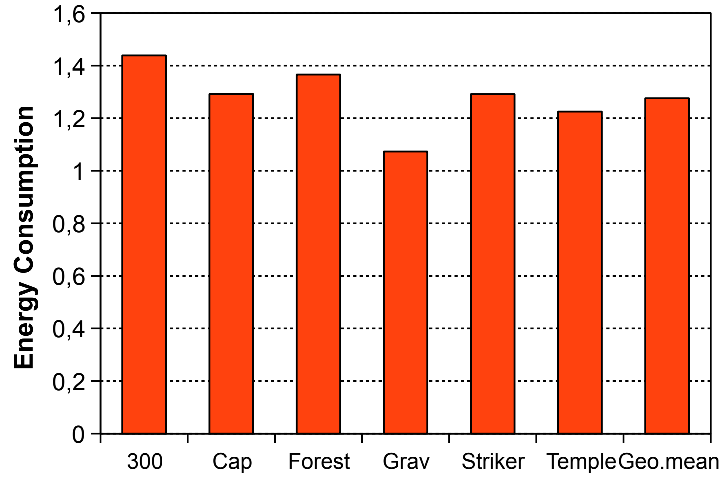


Figure A.14: Energy consumption of Z pre-pass normalized to the baseline IMR GPU.

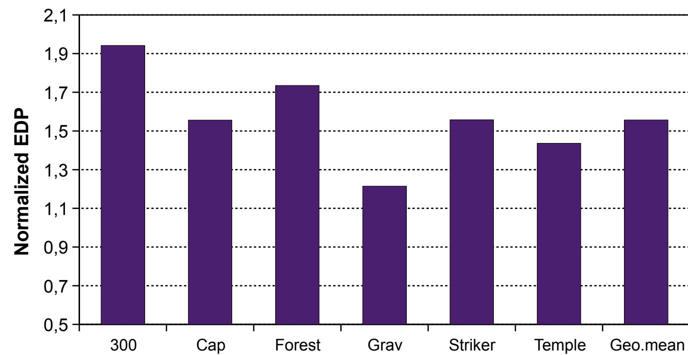


Figure A.15: Energy Delay Product of Z pre-pass normalized to the baseline IMR GPU (lower than 1 is better).

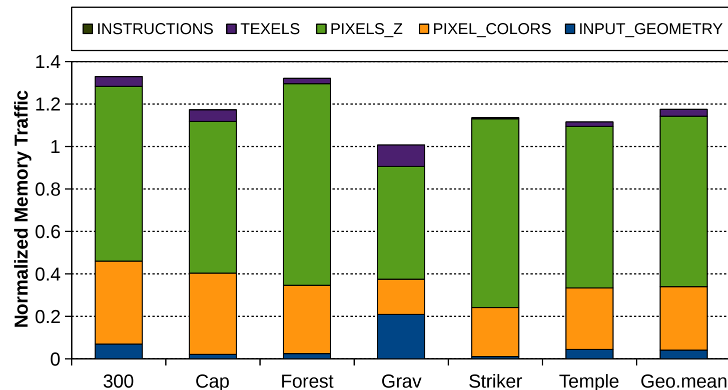


Figure A.16: Memory bandwidth usage of Z pre-pass normalized to baseline IMR GPU.

The normalized EDP of Z pre-pass is much worse than in the baseline in all the cases, being 1.56x on average. According to these results, Z pre-pass would not be a suitable technique to reduce neither execution time nor energy consumption on applications targeted for low-power GPUs because the overhead incurred by the extra rendering pass more than offsets the time and energy savings of the smaller fragment processing. Nevertheless, it makes sense to apply Z pre-pass in a context where the Fragment Processing represents higher computing demands, like in consoles and desktops. Note that the higher rendering resolution or higher computational cost per fragment processed (more complex shaders), the more expensive the rendering in terms of fragment processing.

We have done experiments to compare VRO and Z pre-pass in a scenario with stressed fragment processors. Given that we do not count with the source code of the benchmarks, we exacerbate the cost of the fragment processing by reducing the number of Fragment Processors in the GPU to just one. Doing this, the number of fragments processed per Fragment Processor is incremented eight times. Our goal is to approximate the effects of increasing the resolution from 0.91 Megapixels (1200*768) to 7.4 Megapixels (1200*768*8), which is a rendering resolution that represents the 89% of the 8.3 Megapixels of 4K resolution. Current consoles like XBOX One [75] and PS4 [73] commonly render at 1920x1080 resolution (or lower) and in desktops the rendering resolution can be 3840x2160 (4K).

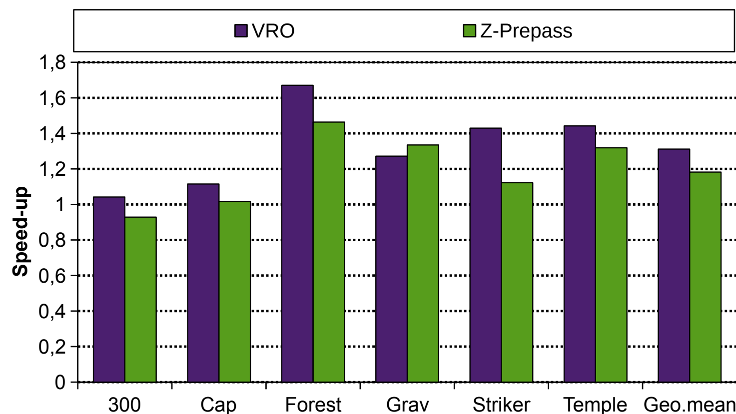


Figure A.17: Speed-up of VRO and Z pre-pass normalized to the baseline IMR GPU (one FP in all cases).

Figure A.17 shows the speedup of both VRO and Z pre-pass when executed in a GPU with just one Fragment Processor. In all the applications but one, the performance of VRO is better than the performance of Z pre-pass. VRO achieves 1.31x speed-up on average while Z pre-pass achieves 1.18x on average. VRO has no slowdown, whereas Z pre-pass introduces slowdown in *300*. Despite Z pre-pass achieves a higher overshading reduction than VRO the extra rendering pass of Z pre-pass greatly penalizes its execution time. Regarding energy, VRO consumes 0.81x and Z pre-pass 1.17x. In *Gravity*, Z pre-pass performs slightly better than VRO in terms of speed-up but in terms of energy Z pre-pass is significantly worse than VRO, because Z pre-pass significantly increases the traffic with main memory. In terms of EDP, VRO is much more efficient than Z pre-pass in all the cases (see Figure A.19), being 0.62x on average while the EDP of Z pre-pass is 0.99x, almost equal to the EDP of the baseline GPU.

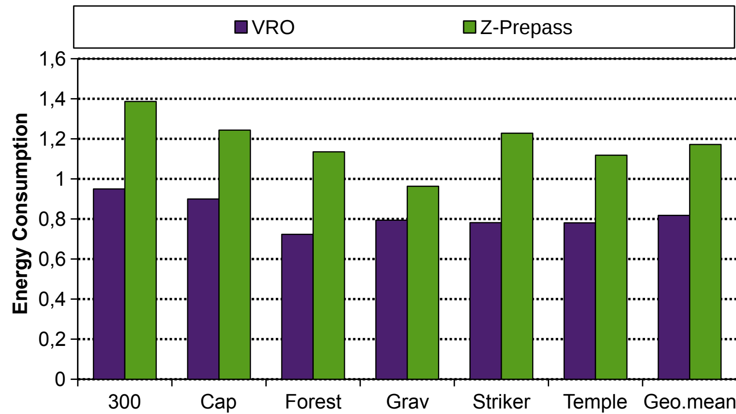


Figure A.18: Energy consumption of VRO and Z pre-pass normalized to the baseline IMR GPU (one FP in all cases).

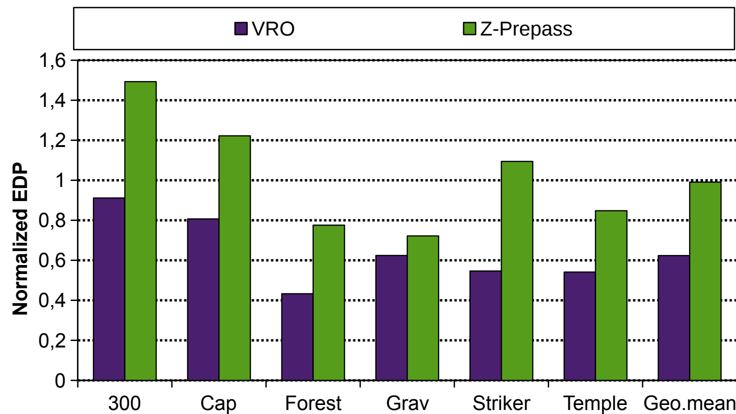


Figure A.19: Energy Delay Product of VRO and Z pre-pass normalized to the baseline IMR GPU (lower than 1 is better, one FP in all cases).

A.4 Conclusions

This subsection presents IMR-VRO, a technique that effectively reduces overshading by culling hidden surfaces. IMR-VRO includes a small hardware unit that stores the order relations among the objects of a scene of the current frame in a buffer. This information is used in the next frame to create a Visibility Rendering Order that guides the Command Processor. The hardware overhead of this technique is minimum, requiring less than 1% of the total area of the GPU, while its latency is vastly offset by the reduction in the execution time of the Graphics Pipeline.

For a set of unmodified commercial applications for Android, IMR-VRO outperforms state-of-the-art techniques in performance and energy consumption by reducing the overshading without the need of an expensive extra rendering pass to determine visibility. IMR-VRO achieves up to 1.32x speed-up (1.16x on average) and down to 0.79x energy consumption (0.85x on average) with respect to the baseline IMR GPU. IMR-VRO is much more efficient than Z pre-pass because most of the

computations required to create the Visibility Rendering Order are reused from the rendering of the previous frame, while Z pre-pass requires an extra render pass that introduces significant overheads.

Bibliography

[1] Falanx mali 110/55 ip cores first to support latest opengl es 1.1 standard. http://www.prnewswire.com/news-releases/falanx-mali-11055-ip-cores-first-to-support-latest-opengl-es-11-standard-54772052.html, 2005.

[2] Designing physics algorithms for gpu architecture. game developers conference. http://www.gdcvault.com/play/1014278/Physics-for-Game, 2011.

[3] Mobile app store downloads worldwide (2010-2016). http://www.gartner.com/newsroom/id/2592315, 2013.

[4] Arm guide to unity enhancing your mobile games: Use depth pre-pass. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.100140_0201_00_en/nic1434707722257.html, 2014.

[5] Movile devides shipments 2013. http://www.gartner.com/newsroom/id/2875017, 2014.

[6] The blacksmith. https://unity3d.com/es/pages/the-blacksmith, 2015.

[7] Movile devides shipments 2014. http://www.gartner.com/newsroom/id/3088221, 2015.

[8] The most used smartphone screen resolutions in 2016. https://deviceatlas.com/blog/most-used-smartphone-screen-resolutions-in-2016, 2016.

[9] Movile devides shipments 2015. http://www.gartner.com/newsroom/id/3468817, 2016.

[10] Android platform architecture. https://developer.android.com/guide/platform/index.html, 2017.

[11] Hardware gpu market. http://hwstats.unity3d.com/mobile/gpu.html, 2017.

[12] ios developer webpage. https://developer.apple.com/develop/, 2017.

[13] Most popular apple app store categories in march 2017. https://www.statista.com/statistics/270291/popular-categories-in-the-app-store/, 2017.

[14] Movile devides shipments 2016. http://www.gartner.com/newsroom/id/3560517, 2017.

[15] Number of apps available in leading app stores as of march 2017. https://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/, 2017.

[16] Opengl es 1.x. https://www.khronos.org/api/opengles/1_X, 2017.

[17] Opengl es 3.x. https://www.khronos.org/api/opengles/3_X, 2017.

[18] Opengl es conformant products. https://www.khronos.org/conformance/adopters/conformant-products#opengles, 2017.

[19] Operating system market share worldwide (june 2012 to june 2017). http://gs.statcounter.com/os-market-share#monthly-201206-201706, 2017.

[20] Renderdoc. https://renderdoc.org, 2017.

[21] Unity 3d game engine. https://en.wikipedia.org/wiki/Unity_(game_engine), 2017.

[22] Unreal engine. https://docs.unrealengine.com/latest/INT/Platforms/Mobile/index.html, 2017.

[23] Vivante vega 3d technology. http://www.vivantecorp.com/index.php/en/technology/3d.html, 2017.

[24] Cobra 3d model by alexander bruckner. https://free3d.com/3d-model/ac-cobra-269-83668.html, accessed August 15, 2017.

[25] Line clipping. https://en.wikipedia.org/wiki/Cohen%E2%80%93Sutherland_algorithm, accessed August 15, 2017.

[26] Painter's algorithm. https://en.wikipedia.org/wiki/Painter%27s_algorithm, accessed August 15, 2017.

[27] Polygon clipping. https://en.wikipedia.org/wiki/Sutherland%E2%80%93Hodgman_algorithm, accessed August 15, 2017.

[28] Z-buffer algorithm'. https://en.wikipedia.org/wiki/Z-buffering#Z-Culling, accessed August 15, 2017.

[29] Sweep and prune. https://en.wikipedia.org/wiki/Sweep_and_prune, accessed August 29, 2017.

[30] Qualcomm snapdragon s4 (krait) performance preview. http://www.anandtech.com/show/5559/qualcomm-snapdragon-s4-krait-performance-preview-msm8960-adreno-225-benchmarks/4, accessed July 10, 2017.

[31] J.d. power ratings. smartphone battery life has become a significant drain on customer satisfaction and loyalty. http://www.jdpower.com/press-releases/2012-us-wireless-smartphone-and-traditional-mobile-phone-satisfaction-studies-volume, accessed July 15, 2017.

[32] J.d. power ratings. wireless charging and fingerprint scanner technology amp up smartphone user satisfaction, says j.d. power study. http://www.jdpower.com/press-releases/2016-us-wireless-smartphone-satisfaction-study-volume-1, accessed July 15, 2017.

[33] Qualcomm. power performance white paper. when mobile apps use too much power. https://developer.qualcomm.com/software/trepn-power-profiler, accessed July 15, 2017.

[34] Snapdragon 410 processor. https://www.qualcomm.com/products/snapdragon/processors/410, accessed July 15, 2017.

[35] Trepn power profiler. https://developer.qualcomm.com/software/trepn-power-profiler, accessed July 15, 2017.

[36] Antutu benchmark. http://www.antutu.com/en/index.htm, accessed July, 2017.

[37] Adreno gpus. https://en.wikipedia.org/wiki/Adreno, accessed June, 2017.

[38] Adreno gpus. https://developer.qualcomm.com/software/adreno-gpu-sdk/gpu, accessed June, 2017.

[39] Apple a10. https://en.wikipedia.org/wiki/Apple_A10, accessed June, 2017.

[40] Apple a8. https://en.wikipedia.org/wiki/Apple_A8, accessed June, 2017.

[41] Apple a9. https://en.wikipedia.org/wiki/Apple_A9, accessed June, 2017.

[42] Directx. https://en.wikipedia.org/wiki/DirectX, accessed June, 2017.

[43] Flexrender. https://www.qualcomm.com/videos/flexrender, accessed June, 2017.

[44] Google play store. https://play.google.com/store, accessed June, 2017.

[45] Helio x10. http://mediatek-helio.com/x10/, accessed June, 2017.

[46] Helio x30. https://www.mediatek.com/products/smartphones/mediatek-helio-x30, accessed June, 2017.

[47] Htc one m9. https://en.wikipedia.org/wiki/HTC_One_M9, accessed June, 2017.

[48] Ipad. https://en.wikipedia.org/wiki/IPad, accessed June, 2017.

[49] Iphone 6. https://en.wikipedia.org/wiki/IPhone_6, accessed June, 2017.

[50] Iphone 7. https://en.wikipedia.org/wiki/IPhone_7, accessed June, 2017.

[51] itunes. https://itunes.apple.com/us/genre/ios/id36?mt=8, accessed June, 2017.

[52] Khronos group. https://www.khronos.org/about, accessed June, 2017.

[53] Mali 200 gpu. https://www.arm.com/products/mali-200.php, accessed June, 2017.

[54] Mali 400 mp series gpu. https://developer.arm.com/products/graphics-and-multimedia/mali-gpus/mali-400-mp-series-gpu, accessed June, 2017.

[55] Mali graphics processing from arm. http://www.arm.com/products/graphics-and-multimedia/mali-gpu, accessed June, 2017.

[56] Nintendo switch. https://en.wikipedia.org/wiki/Nintendo_Switch, accessed June, 2017.

[57] Opengl es. https://www.khronos.org/opengles/, accessed June, 2017.

[58] Opengl es 2.x. https://www.khronos.org/opengles/2_X/, accessed June, 2017.

[59] Opengl es overview. http://en.wikipedia.org/wiki/OpenGL_ES, accessed June, 2017.

[60] Powervr gpus. https://en.wikipedia.org/wiki/PowerVR, accessed June, 2017.

[61] Powervr gpus. https://en.wikipedia.org/wiki/PowerVR, accessed June, 2017.

[62] Psvita. https://en.wikipedia.org/wiki/PlayStation_Vita, accessed June, 2017.

[63] Samsung exynos soc. http://www.samsung.com/semiconductor/minisite/Exynos/w/, accessed June, 2017.

[64] Samsung galaxy s5. https://en.wikipedia.org/wiki/Samsung_Galaxy_S5, accessed June, 2017.

[65] Samsung galaxy s8. https://en.wikipedia.org/wiki/Samsung_Galaxy_S8, accessed June, 2017.

[66] Shield tv. https://www.nvidia.es/shield/shield-tv/, accessed June, 2017.

[67] Snapdragon 600. https://www.qualcomm.com/products/snapdragon/processors/600, accessed June, 2017.

[68] Snapdragon msm8x55. https://www.qualcomm.com/media/documents/files/snapdragon-msm8x55-apq8055-product-brief.pdf, accessed June, 2017.

[69] Snapdragon socs. https://www.qualcomm.com/products/snapdragon, accessed June, 2017.

[70] Tegra processors. http://www.nvidia.com/object/tegra.html, accessed June, 2017.

[71] Tegra x1 white paper. http://international.download.nvidia.com/pdf/tegra/Tegra-X1-whitepaper-v1.0.pdf, accessed June, 2017.

[72] Vega powered products. http://www.vivantecorp.com/index.php/en/products/powered-by-vivante.html, accessed June, 2017.

[73] Playstation 4 technical specifications. https://en.wikipedia.org/wiki/PlayStation_4_technical_specifications, accessed November 9, 2017.

[74] Qemu: the fast! processor emulator. https://www.qemu.org, accessed November 9, 2017.

[75] Xbox one. https://en.wikipedia.org/wiki/Xbox_One, accessed November 9, 2017.

[76] Common reasons users uninstall mobile apps. http://www.gummicube.com/blog/2017/08/common-reasons-users-uninstall-mobile-apps, accessed October 16, 2017.

[77] A comprehensive survey of 3,000+ mobile app users. https://techbeacon.com/resources/survey-mobile-app-users-report-failing-meet-user-expectations, accessed October 16, 2017.

[78] Top 12 reasons why users frequently uninstall mobile apps. https://www.linkedin.com/pulse/top-12-reasons-why-users-frequently-uninstall-mobile-apps-fakhruddin/, accessed October 16, 2017.

[79] Why do users install and delete apps? a survey study. http://www.gonzalez-huerta.net/wp-content/uploads/2017/06/ICSOB17.pdf, accessed October 16, 2017.

[80] Why users uninstall apps. https://software.intel.com/en-us/blogs/2013/11/14/why-users-uninstall-apps, accessed October 16, 2017.

[81] Netflix. https://www.netflix.com, accessed October, 2017.

[82] Rasterization: a practical implementation. https://www.scratchapixel.com/lessons/3d-basic-rendering/rasterization-practical-implementation/rasterization-stage, accessed October, 2017.

[83] Software rasterization algorithms for filling triangles. http://www.sunshine2k.de/coding/java/TriangleRasterization/TriangleRasterization.html, accessed October, 2017.

[84] Tg shader infrastructure. https://www.freedesktop.org/wiki/Software/gallium/tgsi-specification.pdf, accessed October, 2017.

[85] Thermal throttling. which soc's are the worst offenders? https://www.mobiledroid.co.uk/blog/thermal-throttling-which-socs-are-worst/, accessed October, 2017.

[86] Amd comments on gpu stuttering. http://www.anandtech.com/show/6857/amd-stuttering-issues-driver-roadmap-fraps/2, accessed October 29, 2016.

[87] Android studio. https://developer.android.com/studio/intro/index.html, accessed October 29, 2016.

[88] Gallium3d. https://www.freedesktop.org/wiki/Software/gallium/, accessed October 29, 2016.

[89] Mali-400 mp: A scalable gpu for mobile devices. http://www.highperformancegraphics.org/previous/www_2010/media/Hot3D/HPG2010_Hot3D_ARM.pdf, accessed October 29, 2016.

[90] The mali gpu: An abstract machine, part 1 - frame pipelining. https://community.arm.com/groups/arm-mali-graphics/blog/2014/02/03, accessed October 29, 2016.

[91] Powervr. http://www.imgtec.com/powervr/powervr-architecture.asp, accessed October 29, 2016.

[92] Stuttering in game graphics: Detection and solutions. https://developer.nvidia.com/sites/default/files/akamai/gameworks/CN/Stuttering_Analysis_EN.pdf, accessed October 29, 2016.

[93] Tegra 4 white paper. http://www.nvidia.com/docs/IO/116757/Tegra_4_GPU_Whitepaper_FINALv2.pdf, accessed October 29, 2016.

[94] Arm cortex a9. http://arm.com/products/processors/cortex-a/cortex-a9.php, accessed September 9, 2014.

[95] Bullet projects. http://en.wikipedia.org/wiki/Bullet_%28software%29, accessed September 9, 2014.

[96] Gfxbench. https://gfxbench.com/benchmark.jsp, accessed September 9, 2017.

[97] The khronos group inc. opengl es 3.0.5 specification. https://www.khronos.org/registry/OpenGL/specs/es/3.0/es_spec_3.0.pdf, accessed September 9, 2017.

[98] The khronos group inc. opengl es 3.1 specification. https://www.khronos.org/registry/OpenGL/specs/es/3.1/es_spec_3.1.pdf, accessed September 9, 2017.

[99] The khronos group inc. opengl es 3.2 specification. https://www.khronos.org/registry/OpenGL/specs/es/3.2/es_spec_3.2.pdf, accessed September 9, 2017.

[100] The khronos group inc. opengl es common profile specification version 2.0.25. http://www.khronos.org/registry/gles/specs/2.0/es_full_spec_2.0.25.pdf, accessed September 9, 2017.

[101] The khronos group inc. opengl es common/common-lite profile specification version 1.1.12. http://www.khronos.org/registry/gles/specs/1.1/es_full_spec_1.1.12.pdf, accessed September 9, 2017.

[102] T. Akenine-Moller and J. Strom. Graphics processing units for handhelds. *Proceedings of the IEEE*, 96(5):779–789, May 2008.

[103] T. Akenine-Moller and J. Strom. Graphics processing units for handhelds. *Proc. of the IEEE*, 96(5):779–789, May 2008.

[104] Tomas Akenine-Möller, Eric Haines, and Natty Hoffman. *Real-Time Rendering 3rd Edition*. A. K. Peters, Ltd., Natick, MA, USA, 2008.

[105] Tomas Akenine-Möller and Jacob Ström. Graphics for the masses: A hardware rasterization architecture for mobile phones. *ACM Trans. Graph.*, 22(3):801–808, July 2003.

[106] I. Antochi. *Suitability of Tile-based Rendering for Low-power 3d Graphics Accelerators*. Universitatea Politehnica Bucureşti, 2007.

[107] Sigal Ar, Bernard Chazelle, and Ayellet Tal. Self-customized bsp trees for collision detection. *Comput. Geom. Theory Appl.*, 15(1-3):91–102, February 2000.

[108] J. Arnau. *Energy-Efficient Mobile GPU Systems*. PhD thesis, Universitat Politècnica de Catalunya, Apr 2015.

[109] J.-M. Arnau, J.-M. Parcerisa, and P. Xekalakis. Boosting mobile gpu performance with a decoupled access/execute fragment processor. In *Comp. Archit. (ISCA), 2012 39th Annual Int. Symp. on*, pages 84–93, June 2012.

[110] 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*, PACT '13, pages 83–92, Piscataway, NJ, USA, 2013. IEEE Press.

[111] Jose-Maria Arnau, Joan-Manuel Parcerisa, and Polychronis Xekalakis. Teapot: A toolset for evaluating performance, power and image quality on mobile graphics systems. In *Proc. of the 27th Int. ACM Conf. on Int. Conf. on Supercomputing*, ICS '13, pages 37–46, New York, NY, USA, 2013. ACM.

[112] Jose-Maria Arnau, Joan-Manuel Parcerisa, and Polychronis Xekalakis. Eliminating redundant fragment shader executions on a mobile gpu via hardware memoization. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, pages 529–540, Piscataway, NJ, USA, 2014. IEEE Press.

[113] G. Baciu and Wingo Sai-Keung Wong. Rendering in object interference detection on conventional graphics workstations. In *Proc. of the 5th Pacific Conf. on Comp. Graphics and Applications*, PG '97, pages 51–, Washington, DC, USA, 1997. IEEE Comp. Society.

[114] George Baciu and Wingo S. K. Wong. Image-based techniques in a hybrid collision detector. *IEEE Transactions on Visualization and Comp. Graphics*, 9(2):254–271, April 2003.

[115] George Baciu and WingoSai-Keung Wong. Image-based collision detection. In DavidD. Zhang, Mohamed Kamel, and George Baciu, editors, *Integrated Image and Graphics Technologies*, volume 762 of *The Int. Series in Engineering and Comp. Science*, pages 75–94. Springer US, 2004.

[116] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt. Analyzing cuda workloads using a detailed gpu simulator. In *2009 IEEE International Symposium on Performance Analysis of Systems and Software*, pages 163–174, April 2009.

[117] Dirk Bartz, Michael Meißner, and Tobias Hüttner. Opengl-assisted occlusion culling for large polygonal models, 1999.

[118] Jiri Bittner, Michael Wimmer, Harald Piringer, and Werner Purgathofer. Coherent Hierarchical Culling: Hardware Occlusion Queries Made Useful. *Computer Graphics Forum*, 2004.

[119] Pierre Charbit, Stéphan Thomassé, and Anders Yeo. The minimum feedback arc set problem is np-hard for tournaments. *Comb. Probab. Comput.*, 16(1):1–4, January 2007.

[120] Niladrish Chatterjee, Mike O'Connor, Donghyuk Lee, Daniel R. Johnson, Stephen W. Keckler, Minsoo Rhu, and William J. Dally. Architecting an energy-efficient dram system for gpus. In *23rd International Symposium on Higher Performance Computer Architecture*, HPCA, 2017.

[121] Wei Chen, Huagen Wan, Hongxin Zhang, Hujun Bao, and Qunsheng Peng. Interactive collision detection for complex and deformable models using programmable graphics hardware. In *Proceedings of the ACM Symposium on Virtual Reality Software and Technology*, VRST '04, pages 10–15, New York, NY, USA, 2004. ACM.

[122] Xiang Chen, Yiran Chen, Zhan Ma, and Felix C. A. Fernandes. How is energy consumed in smartphone display applications? In *Proceedings of the 14th Workshop on Mobile Computing Systems and Applications*, HotMobile '13, pages 3:1–3:6, New York, NY, USA, 2013. ACM.

[123] Slo-Li Chu, Chih-Chieh Hsiao, and Chiu-Cheng Hsieh. An energy-efficient unified register file for mobile gpus. In *Embedded and Ubiquitous Computing (EUC), 2011 IFIP 9th Int. Conf. on*, pages 166–173, Oct 2011.

[124] Petrik Clarberg, Robert Toth, and Jacob Munkberg. A sort-based deferred shading architecture for decoupled sampling. *ACM Trans. Graph.*, 32(4):141:1–141:10, July 2013.

[125] D. Cohen-Or, Y. L. Chrysanthou, C. T. Silva, and F. Durand. A survey of visibility for walkthrough applications. *IEEE Transactions on Visualization and Computer Graphics*, 9(3):412–431, July 2003.

[126] S. Collange, M. Daumas, D. Defour, and D. Parello. Barra: A parallel functional simulator for gpgpu. In *2010 IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, pages 351–360, Aug 2010.

[127] Qt Community. Threads Events QObjects. https://wiki.qt.io/Threads_Events_QObjects, 2015. [Online; accessed 10-August-2017].

[128] Shane Cook. *CUDA Programming: A Developer's Guide to Parallel Computing with GPUs*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2013.

[129] E. Coumans. Bullet physics library. https://code.google.com/p/bullet/downloads/list, 2013.

[130] Patrick Cozzi and Christophe Riccio. *OpenGL Insights pp. 396, 417, 494*. CRC Press, July 2012. http://www.openglinsights.com/.

[131] Enrique de Lucas, Pedro Marcuello, Joan-Manuel Parcerisa, and Antonio González. Ultra-low power render-based collision detection for cpu/gpu systems. In *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO-48, pages 445–456, New York, NY, USA, 2015. ACM.

[132] Michael Deering, Stephanie Winner, Bic Schediwy, Chris Duffy, and Neil Hunt. The triangle processor and normal vector shader: A vlsi system for high performance graphics. *SIGGRAPH Comput. Graph.*, 22(4):21–30, June 1988.

[133] V. M. del Barrio, C. Gonzalez, J. Roca, A. Fernandez, and Espasa E. Attila: a cycle-level execution-driven simulator for modern gpu architectures. In *2006 IEEE International Symposium on Performance Analysis of Systems and Software*, pages 231–241, March 2006.

[134] Peng Du, Elvis S. Liu, and Toyotaro Suzumura. Parallel continuous collision detection for high-performance gpu cluster. In *Proceedings of the 21st ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, I3D '17, pages 4:1–4:7, New York, NY, USA, 2017. ACM.

[135] Christer Ericson. *Real-Time Collision Detection (The Morgan Kaufmann Series in Interactive 3-D Technology) (The Morgan Kaufmann Series in Interactive 3D Technology)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004.

[136] WenShan Fan, Bin Wang, Jean-Claude Paul, and JiaGuang Sun. An octree-based proxy for collision detection in large-scale particle systems. *Science China Information Sciences*, 56(1):1–10, Jan 2013.

[137] Kayvon Fatahalian, Solomon Boulos, James Hegarty, Kurt Akeley, William R. Mark, Henry Moreton, and Pat Hanrahan. Reducing shading on gpus using quad-fragment merging. *ACM Trans. Graph.*, 29(4):67:1–67:8, July 2010.

[138] François Faure, Sébastien Barbier, Jérémie Allard, and Florent Falipou. Image-based collision detection and response between arbitrary volume objects. In *Proc. of the 2008 ACM SIGGRAPH/Eurographics Symp. on Comp. Animation*, SCA '08, pages 155–162, Aire-la-Ville, Switzerland, Switzerland, 2008. Eurographics Association.

[139] W. W. L. Fung, I. Sham, G. Yuan, and T. M. Aamodt. Dynamic warp formation and scheduling for efficient gpu control flow. In *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*, pages 407–420, Dec 2007.

[140] E.G. Gilbert, D.W. Johnson, and S.S. Keerthi. A fast procedure for computing the distance between complex objects in three-dimensional space. *Robotics and Automation, IEEE Journal of*, 4(2):193–203, Apr 1988.

[141] 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*, I3D '05, pages 49–56, New York, NY, USA, 2005. ACM.

[142] Naga K. Govindaraju, Stephane Redon, Ming C. Lin, and Dinesh Manocha. Cullide: Interactive collision detection between complex models in large environments using graphics hardware. In *ACM SIGGRAPH 2005 Courses*, SIGGRAPH '05, New York, NY, USA, 2005. ACM.

[143] N.K. Govindaraju, M.C. Lin, and D. Manocha. Fast and reliable collision culling using graphics hardware. *Visualization and Comp. Graphics, IEEE Transactions on*, 12(2):143–154, March 2006.

[144] Jason Gregory. *Game Engine Architecture, Second Edition*. A. K. Peters, Ltd., Natick, MA, USA, 2nd edition, 2014.

[145] Eric Haines and Steven Worley. Fast, low memory z-buffering when performing medium-quality rendering. *J. Graph. Tools*, 1(3):1–6, February 1996.

[146] 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*, I3D '16, pages 23–27, New York, NY, USA, 2016. ACM.

[147] Tianyi David Han and Tarek S. Abdelrahman. Reducing branch divergence in gpu programs. In *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-4, pages 3:1–3:8, New York, NY, USA, 2011. ACM.

[148] Jon Hasselgren and Tomas Akenine-Möller. Efficient depth buffer compression. In *Proc. of the 21st ACM SIGGRAPH/EUROGRAPHICS Symp. on Graphics Hardware*, GH '06, pages 103–110, New York, NY, USA, 2006. ACM.

[149] Liang He, Ricardo Ortiz, Andinet Enquobahrie, and Dinesh Manocha. Interactive continuous collision detection for topology changing models using dynamic clustering. In *Proceedings of the 19th Symposium on Interactive 3D Graphics and Games*, i3D '15, pages 47–54, New York, NY, USA, 2015. ACM.

[150] Songtao He, Yunxin Liu, and Hucheng Zhou. Optimizing smartphone power consumption through dynamic resolution scaling. In *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking*, MobiCom '15, pages 27–39, New York, NY, USA, 2015. ACM.

[151] Bruno Heidelberger, Matthias Teschner, and Markus H. Gross. Real-time volumetric intersections of deforming objects. In *Proc. of the Vision, Modeling, and Visualization Conf. 2003 (VMV 2003), München, Germany, November 19-21, 2003*, pages 461–468, 2003.

[152] Bruno Heidelberger, Matthias Teschner, and Markus H. Gross. Detection of collisions and self-collisions using image-space techniques. In *The 12-th Int. Conf. in Central Europe on Comp. Graphics, Visualization and Comp. Vision'2004*, pages 145–152, 2004.

[153] Sunpyo Hong and Hyesoon Kim. An integrated gpu power and performance model. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, pages 280–289, New York, NY, USA, 2010. ACM.

[154] Pablo Jimenez, Federico Thomas, and Carme Torras. 3d collision detection: A survey. 25:269–285, 04 2001.

[155] B. Juurlink, I. Antochi, D. Crisu, S. Cotofana, and S. Vassiliadis. Graal: A framework for low-power 3d graphics accelerators. *IEEE Computer Graphics and Applications*, 28(4):63–73, July 2008.

[156] A. B. Kahn. Topological sorting of large networks. *Commun. ACM*, 5(11):558–562, November 1962.

[157] Dave Knott and Dinesh K. Pai. Cinder: Collision and interference detection in real-time using graphics hardware, 2003.

[158] S. Kockara, T. Halic, K. Iqbal, C. Bayrak, and R. Rowe. Collision detection: A survey. In *Systems, Man and Cybernetics, 2007. ISIC. IEEE Int. Conf. on*, pages 4046–4051, Oct 2007.

[159] James H. Laros III, Kevin Pedretti, Suzanne M. Kelly, Wei Shu, Kurt Ferreira, John Vandyke, and Courtenay Vaughan. *Energy Delay Product*, pages 51–55. Springer London, London, 2013.

[160] Orion Sky Lawlor and Laxmikant V. Kalée. A voxel-based parallel collision detection algorithm. In *Proceedings of the 16th International Conference on Supercomputing*, ICS '02, pages 285–293, New York, NY, USA, 2002. ACM.

[161] Victor W. Lee, Changkyu Kim, Jatin Chhugani, Michael Deisher, Daehyun Kim, Anthony D. Nguyen, Nadathur Satish, Mikhail Smelyanskiy, Srinivas Chennupaty, Per Hammarlund, Ronak Singhal, and Pradeep Dubey. Debunking the 100x gpu vs. cpu myth: An evaluation of throughput computing on cpu and gpu. In *Proc. of the 37th Annual Int. Symp. on Comp. Archit.*, ISCA '10, pages 451–460, New York, NY, USA, 2010. ACM.

[162] Jingwen Leng, Tayler Hetherington, Ahmed ElTantawy, Syed Gilani, Nam Sung Kim, Tor M. Aamodt, and Vijay Janapa Reddi. Gpuwattch: Enabling energy optimizations in gpgpus. *SIGARCH Comput. Archit. News*, 41(3):487–498, June 2013.

[163] Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. The mcpat framework for multicore and manycore archit.s: Simultaneously modeling power, area, and timing. *ACM Trans. Archit. Code Optim.*, 10(1):5:1–5:29, April 2013.

[164] Gábor Liktor and Carsten Dachsbacher. Decoupled deferred shading for hardware rasterization. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, I3D '12, pages 143–150, New York, NY, USA, 2012. ACM.

[165] Jieun Lim, Nagesh B. Lakshminarayana, Hyesoon Kim, William Song, Sudhakar Yalamanchili, and Wonyong Sung. Power modeling for gpu architectures using mcpat. *ACM Trans. Des. Autom. Electron. Syst.*, 19(3):26:1–26:24, June 2014.

[166] Steve Marschner and Peter Shirley. *Fundamentals of Computer Graphics, Fourth Edition*. A. K. Peters, Ltd., Natick, MA, USA, 4th edition, 2016.

[167] Microsoft. Primitive Topologies. https://msdn.microsoft.com/en-us/library/windows/desktop/bb205124(v=vs.85).aspx#Primitive_Types, 2017. [Online; accessed 11-August-2017].

[168] Bren Mochocki, Kanishka Lahiri, and Srihari Cadambi. Power analysis of mobile 3d graphics. In *Proc. of the Conf. on Design, Automation and Test in Europe: Proc.*, DATE '06, pages 502–507, 3001 Leuven, Belgium, Belgium, 2006. European Design and Automation Association.

[169] Victor Moya, Carlos Gonzalez, Jordi Roca, Agustin Fernandez, and Roger Espasa. Shader performance analysis on a modern gpu architecture. In *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 38, pages 355–364, Washington, DC, USA, 2005. IEEE Computer Society.

[170] Karol Myszkowski, OlegG. Okunev, and TosiyasuL. Kunii. Fast collision detection between complex solids using rasterizing graphics hardware. *The Visual Comp.*, 11(9):497–511, 1995.

[171] Diego Nehab, Joshua Barczak, and Pedro V. Sander. Triangle order optimization for graphics hardware computation culling. In *Proceedings of the 2006 Symposium on Interactive 3D Graphics and Games*, I3D '06, pages 207–211, New York, NY, USA, 2006. ACM.

[172] Thomas J. Olson. Hardware 3d graphics acceleration for mobile devices. In *2008 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 5344–5347, March 2008.

[173] A. Patel, F. Afram, Shunfei Chen, and K. Ghose. Marss: A full system simulator for multicore x86 cpus. In *Design Automation Conf. (DAC), 2011 48th ACM/EDAC/IEEE*, pages 1050–1055, June 2011.

[174] Shruti Patil, Yeseong Kim, Kunal Korgaonkar, Ibrahim Awwal, and Tajana S. Rosing. *Characterization of User's Behavior Variations for Design of Replayable Mobile Workloads*, pages 51–70. Springer International Publishing, Cham, 2015.

[175] Kévin Perrot and Trung Van Pham. Feedback arc set problem and np-hardness of minimum recurrent configuration problem of chip-firing game on directed graphs. *Annals of Combinatorics*, 19(2):373–396, 2015.

[176] J. Pool. *Energy-Precision Tradeoffs in the Graphics Pipeline*. dissertation, The University of North Caroline at Chapel Hill, 2012.

[177] J. Pool, A. Lastra, and M. Singh. An energy model for graphics processing units. In *2010 IEEE International Conference on Computer Design*, pages 409–416, Oct 2010.

[178] Alok Prakash, Hussam Amrouch, Shafique Muhammad, Tulika Mitra, and JÃ¶rg Henkel. Improving mobile gaming performance through cooperative cpu-gpu thermal management, 06 2016.

[179] Jonathan Ragan-Kelley, Jaakko Lehtinen, Jiawen Chen, Michael Doggett, and Frédo Durand. Decoupled sampling for graphics pipelines. *ACM Trans. Graph.*, 30(3):17:1–17:17, May 2011.

[180] Jim Rasmusson, Jon Hasselgren, and Tomas Akenine-Möller. Exact and error-bounded approximate color buffer compression and decompression. In *Proc. of the 22Nd ACM SIGGRAPH/EUROGRAPHICS Symp. on Graphics Hardware*, GH '07, pages 41–48, Aire-la-Ville, Switzerland, Switzerland, 2007. Eurographics Association.

[181] R.Nystrom. Game programming patterns, sequencing patterns: Game loop. http://gameprogrammingpatterns.com/game-loop.html, 2014.

[182] P. Rosenfeld, E. Cooper-Balis, and B. Jacob. Dramsim2: A cycle accurate memory system simulator. *IEEE Computer Architecture Letters*, 10(1):16–19, Jan 2011.

[183] Jarek Rossignac, Abe Megahed, and Bengt olaf Schneider. Interactive inspection of solids: cross-sections and interferences. In *In Proc. of ACM Siggraph*, pages 353–360, 1992.

[184] Takafumi Saito and Tokiichiro Takahashi. Comprehensible rendering of 3-d shapes. *SIGGRAPH Comput. Graph.*, 24(4):197–206, September 1990.

[185] Pedro V. Sander, Diego Nehab, and Joshua Barczak. Fast triangle reordering for vertex locality and reduced overdraw. In *ACM SIGGRAPH 2007 Papers*, SIGGRAPH '07, New York, NY, USA, 2007. ACM.

[186] Rahul Sathe and Tomas Akenine-MÃ¶ller. Pixel Merge Unit. In B. Bickel and T. Ritschel, editors, *EG 2015 - Short Papers*. The Eurographics Association, 2015.

[187] Dean Sekulic. Efficient occlusion culling. In Nvidia, editor, *GPU Gems*, pages 487–503. 2004.

[188] J. W. Sheaffer, D. Luebke, and K. Skadron. A flexible simulation framework for graphics architectures. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, HWWS '04, pages 85–94, New York, NY, USA, 2004. ACM.

[189] Mikio Shinya and Marie-Claire Forgue. Interference detection through rasterization. *The Journal of Visualization and Comp. Animation*, 2(4):132–134, 1991.

[190] Edvard SØrgård. Graphics clusters. Graphics Hardware, 2004.

[191] S. Sowerby and B. Lipchak. Ext_debug_marker. https://www.khronos.org/registry/gles/extensions/EXT/EXT_debug_marker.txt, 2013.

[192] John E. Stone, David Gohara, and Guochun Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *IEEE Des. Test*, 12(3):66–73, May 2010.

[193] Rafael Ubal, Julio Sahuquillo, Salvador Petit, and Pedro López. Multi2sim: A simulation framework to evaluate multicore-multithreaded processors. In *19th Symposium on Computer Architecture and High Performance Computing (SBAC-PAD 2007)*, pages 62–68, 2007.

[194] Gino van den Bergen, Gino Van, and Den Bergen. A fast and robust gjk implementation for collision detection of convex objects, 1999.

[195] Guy R. Wagner and William Maltz. Too hot to hold: Determining the cooling limits for handheld devices. In *Advancements in Thermal Management 2013*, 2013.

[196] Ren Weller. *New Geometric Data Structures for Collision Detection and Haptics*. Springer Publishing Company, Incorporated, 2013.

[197] Andrew Wilson, Ketan Mayer-Patel, and Dinesh Manocha. Spatially-encoded far-field representations for interactive walkthroughs. In *Proc. of ACM Multimedia*, pages 348–357, 2001.

[198] Huai Yu Wang and S Liu. A collision detection algorithm using aabb and octree space division. 989-994:2389–2392, 07 2014.

[199] Hui Zeng, Matt Yourst, Kanad Ghose, and Dmitry Ponomarev. Mptlsim: A cycle-accurate, full-system simulator for x86-64 multicore architectures with coherent caches. *SIGARCH Comput. Archit. News*, 37(2):2–9, July 2009.