ZADÁNÍ BAKALÁŘSKÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Profota** Jméno: **Jakub** Osobní číslo: **491830**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačové grafiky a interakce**

Studijní program: **Otevřená informatika**

Specializace: **Počítačové hry a grafika**

II. ÚDAJE K BAKALÁŘSKÉ PRÁCI

Název bakalářské práce:

Implementace zobrazovacího řetězce pomocí GPGPU technik

Název bakalářské práce anglicky:

Rendering pipeline implementation using GPGPU techniques

Pokyny pro vypracování:

Zmapujte a popište architekturu zobrazovacího řetězce v moderních GPU. Navrhněte implementaci základní podoby zobrazovacího řetězce bez využití grafického API. Pro implementaci využijte jazyk CUDA nebo OpenCL. Soustřeďte se na efektivní paralelní rasterizaci trojuhelníků a základní osvětlovací model. Vyhodnoťte rychlost implementace na neujméně pěti scénách různé geometrické složitosti a navrhněte optimalizace úzkých hrdel výpočtu.

Seznam doporučené literatury:

[1] Samuli Laine, Tero Karras. 'High-performance software rasterization on GPUs.' Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics. 2011.
[2] Michael Kenzel, Bernhard Kerbl, Dieter Schmalstieg, and Markus Steinberger. 2018. A high-performance software graphics pipeline architecture for the GPU. ACM Trans. Graph. 37, 4, Article 140 (2018).

Jméno a pracoviště vedoucí(ho) bakalářské práce:

doc. Ing. Jiří Bittner, Ph.D. Katedra počítačové grafiky a interakce

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) bakalářské práce:

Datum zadání bakalářské práce: **17.02.2023** Termín odevzdání bakalářské práce: **26.05.2023**

Platnost zadání bakalářské práce: **22.09.2024**

doc. Ing. Jiří Bittner, Ph.D. podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	prof. Mgr. Petr Páta, Ph.D. podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Student bere na vědomí, že je povinen vypracovat bakalářskou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací.
Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v bakalářské práci.

.

Datum převzetí zadání Podpis studenta

CTU

CZECH TECHNICAL
UNIVERSITY
IN PRAGUE

F3

Faculty of Electrical Engineering
Department of Computer Graphics and Interaction

Bachelor's Thesis

Implementation of Rendering Pipeline Using General-Purpose Computing on Graphics Processing Units

Jakub Profota
Open Informatics

May 2023
Supervisor: doc. Ing. Jiří Bittner, Ph.D.

Acknowledgement / Declaration

I would like to express my gratitude and appreciation to my supervisor, doc. Ing. Jiří Bittner, Ph.D, for his guidance and patience during the work on the project and this thesis.

Many thanks go to my parents and my sister for their support and encouragement throughout my studies.

I thank my partner and my friends for their emotional support.

I declare that I have prepared the submitted thesis independently and that I have listed all information sources used following the Methodological guidelines on the observance of ethical principles in the preparation of the university thesis.

Prague, May 26 2023

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze, 26. května 2023

....

Abstrakt / Abstract

Tato bakalářská práce implementuje jednoduchý zobrazovací řetězec pomocí technik obecných výpočtů na moderních grafických procesorech. Implementace v technologii CUDA se soustředí na efektivní paralelní rasterizaci trojúhelníků a základní osvětlovací model. Práce mapuje a popisuje architekturu zobrazovacího řetězce a grafických čipů.

Klíčová slova: zobrazovací řetězec, rasterizace, obecné výpočty na grafických procesorech, NVIDIA CUDA

Překlad titulu: Implementace zobrazovacího řetězce pomocí technik obecných výpočtů na grafických procesorech

This bachelor thesis implements a basic graphics pipeline using general-purpose computing techniques on modern graphics processing units. The implementation in CUDA focuses on efficient parallel rasterization of triangles and a basic illumination model. The thesis maps and describes the architecture of the rendering pipeline and graphics chips.

Keywords: Rendering Pipeline, Rasterization, General-Purpose Computing on Graphics Processing Units, NVIDIA CUDA

Contents /

Tables /

Chapter 1
Introduction

Graphics processing units (GPUs) have revolutionized the world of computing, transforming how we experience graphics, accelerate computations, and tackle complex problems. Initially designed to handle the intense computational demands of rendering real-time life-like graphics, GPUs have since evolved into the most pervasive parallel processors with unmatched floating-point performance and programmability. With massive parallelism, high memory bandwidth, and unparalleled power efficiency, GPUs have become instrumental in gaming, scientific research, artificial intelligence, data analytics, and more.

Fueled by the insatiable desire for realistic real-time graphics and the ever-increasing spectrum of data-intensive workloads, the single-purpose fixed-function GPU design has evolved into a flexible, accessible, and programmable architecture. The large and massively parallel processor on the graphics chip is heavily optimized for efficient execution of data-parallel workloads such as graphics pipeline, hence the chip's name. Since the invention of the GPU, most of the stages of the graphics pipeline have become programmable through graphics application programming interfaces (APIs) such as Vulkan [15] or OpenGL [16]. Moreover, these graphics APIs and specialized interfaces like CUDA [17] or OpenCL [18] expose programmable cores for general-purpose computing on graphics chips (GPGPU). These APIs provide access to the GPU from shader programs or extensions to standard programming languages. GPGPU techniques allow for the implementation and effective execution of various data-parallel workloads, such as the purely software-based rendering pipeline.

1.1 Related Work

Before the invention of the GPU, rendering was carried out by a software implementation on a central processing unit (CPU), which is now a thing of the past. Omnipresent dedicated graphics hardware delivering significantly better performance has since fulfilled this role. GPUs have become increasingly programmable throughout the years of development, but the gain in programmability has hit a limit. The GPU architecture is primarily optimized for the highly efficient graphics pipeline, and additional freedom would require altering the pipeline structure, which is firmly rooted in the graphics hardware on the GPU. Moreover, the complexity of the traditional pipeline does not allow changes toward more flexible pipelines, leaving some of the classical rendering problems solved inefficiently [2].

1.1.1 FreePipe

One such problem is multi-fragment effects, such as order-independent transparency of rendered geometry, which would require programmability of the hardware-implemented blending stage. In the traditional graphics pipeline, transparency cannot be solved in a single render pass. FreePipe [1], published in 2010, pioneered real-time software-based rendering using Compute Unified Device Architecture

(CUDA), the world's first specialized GPGPU platform introduced by NVIDIA. FreePipe demonstrates a fully programmable rendering pipeline addressing the issue of order-independent transparency. The implementation introduces two solutions to per-pixel fragment depth sorting in a single geometry pass, modifying the traditional pipeline and thus breaking the constraints imposed by current graphics APIs.

1.1.2 CUDAraster

Developed and published by NVIDIA in 2012 as an experiment to compare the actual performance difference between software and hardware implementations, CUDAraster [2] is currently the most performant completely software-based graphics pipeline on a GPU, obeying the constraints of graphics APIs. Implemented in CUDA and relying on low-level hardware-aware optimizations, CUDAraster focuses primarily on effective rasterization, the usual bottleneck of graphics pipelines. Unlike FreePipe's triangle-parallel approach, with each thread processing one input triangle into many fragments, which are then depth sorted in the programmable blending stage, CUDAraster uses a pixel-parallel approach. The pipeline tiles the screen, then sorts input triangles into tiles by their screen coverage. Each thread processes one pixel in each tile with an assigned queue of triangles. Such an approach does not require a screen-wide depth buffer, which is otherwise exceptionally efficiently implemented in hardware and is thus unparalleled by software implementation.

1.1.3 cuRE

The CUDAraster graphics pipeline is implemented as a sequence of stages, or in terms of CUDA GPGPU programming, as a series of kernel launches. Such an approach can lead to excessive memory consumption since all the data have to be buffered between pipeline stages. It may also suboptimally utilize the GPU hardware with some programmable cores without a workload. cuRE [3], published in 2018, addresses this issue by introducing a fully parallel, multi-stage streaming pipeline design. Instead of launching a kernel per each stage, cuRE launches a single mega-kernel that dynamically load-balances the data flow between the logical pipeline stages. Such an approach is capable of operating efficiently even within bounded memory.

1.2 Thesis Structure

This thesis aims to implement its own entirely software-based rendering pipeline, focusing primarily on effective rasterization. The text also introduces the hardware of the graphics processing units, trying to put the design of the modern chips into a historical perspective. In the second chapter, the thesis introduces graphics processing units and a brief history of NVIDIA GPU chips. The third chapter describes general-purpose computing on the GPU, focusing on CUDA technology. The purpose and the hardware implementation of the graphics pipeline are presented in the fourth chapter, with the fifth chapter diving into an extensive description of the thesis software-based rendering pipeline implementation. The evaluation is provided and discussed in the sixth chapter. A user manual and a brief source code documentation can be found in the appendices.

Chapter 2
Graphics Processing Unit

GPU is a specialized co-processor that offloads computationally intensive data-parallel tasks from a CPU. The original purpose of the graphics chip was to accelerate real-time graphics rendering. However, the substantial increase in programmability and flexibility allowed researchers and developers to advance in various computation-intensive workloads, such as data analytics, physics simulation, or artificial intelligence. The scalable architecture of modern chips allows for massive arrays of GPUs to empower today's most performant cloud and data centers. Moreover, the advancements in GPGPU techniques fueled the exploration of new real-time graphics algorithms, changing the traditional rendering pipeline. Modern graphics cards enable new lighting and shadowing techniques, for example, by facilitating real-time ray tracing.

2.1 Chip Design

The current multi-core architecture of modern processors stems from the physical limitations of the chip manufacturing process. Hardware designers deployed various techniques and technological innovations to bypass the limitations of single-core chips and continue the exponential growth of computing speed with parallelism. However, effective utilization of the performance of the current multi-core architectures requires an explicit redesign of traditional deterministic sequential algorithms.

2.1.1 Manufacturing Process

The processor, the main building block of computers, is the result of a manufacturing process that starts with a crystalline silicon ingot, a heavily refined and purified rod made out of a substance found in sand. Tending the rods requires massive amounts of crystal-clear water and a perfectly sterilized retained environment. Any tiny microscopic impurity may cause the whole manufacturing process to fail. The rod is sliced into thin wafers, which go through a series of processing steps, during which patterns of chemicals are placed on each wafer, creating transistors, conductors, and insulators[1]. The transistor is a switch that conducts or insulates electricity under particular conditions. The combination of billions forms a computation logic [5].

During the processing steps, microscopic impurities of the water itself or imperfections in one of the chemical patterning steps can result in that area of the wafer failing. Defective dies are discarded when the round wafer is diced into square dies or chips and tested for correct behavior. Chips are then connected to the input-output pins of a package through a process of bonding, tested for the second time, and finally shipped to customers. The cost of the manufacturing process and the final product rises quickly as the die size increases due to the lower yield of dies without defects and the smaller number of dies that fit on the wafer [5].

[1] Illustrative video of the processor manufacturing process can be found in NVIDIA's GTC 2023 Keynote at 22:32 [19].

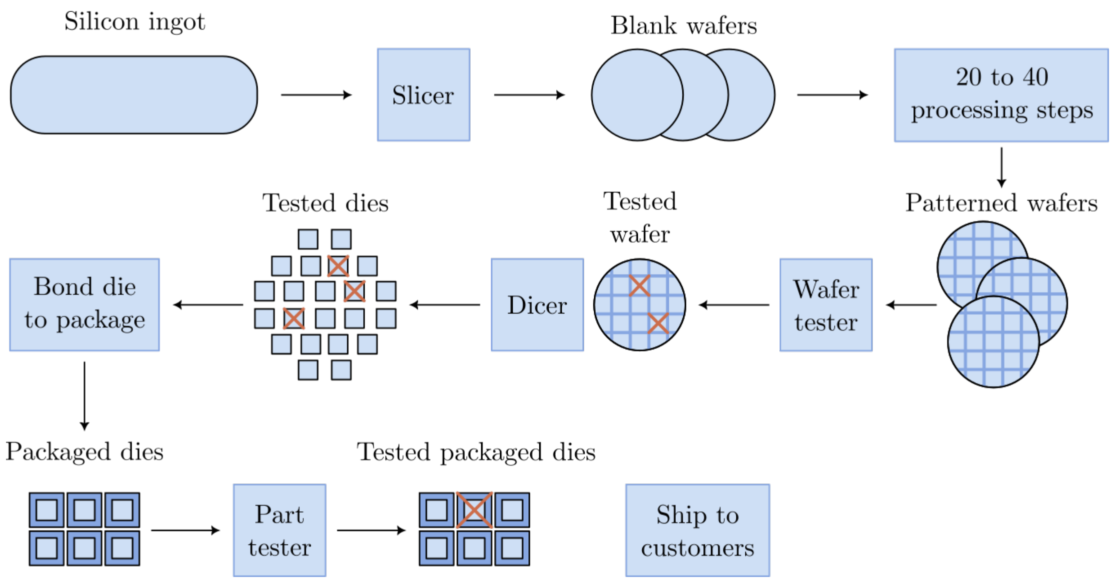


Figure 2.1. Chip manufacturing process [5].

A solution to the issue would be to increase the size of the wafer. However, both the wafer size and the die size have physical limitations. Increasing the diameter of the wafer requires more precise and expensive purifying of the rod and more delicate cutting techniques, as the slicing of the ingot may fail due to the fragile silicon. Increasing the die size restricts the maximum frequency due to the limited speed of electrical impulse transmissions. Supposing a frequency of 3 GHz and the speed of light in a vacuum of $3 \cdot 10^8 ms^{-1}$, the light traverses a distance of 10 centimeters per one tick. Considering the slower speed of electrical impulses compared to the speed of light and additional delays caused by logic circuits, the die size is significantly limited [6].

To reduce the cost and increase the number of transistors, new manufacturing processes, such as TSMC's 3-nanometer technology [20], are used to reduce the size of transistors, increasing the logic density and the speed of the chip at the same power and transistor count. Such a process allows fitting more equally capable chips on a single wafer, offering higher yields. However, by increasing the number of transistors and the frequency, the power consumed rises proportionally, thus generating heat on a much smaller die surface area. Excessive heat must be cooled with increasingly more challenging cooling solutions [6]. Chip designers responded by lowering the operating voltages of circuits, currently running with as low as 1.29 volts, but further decrease causes system instabilities. High temperatures led to a stall of clock frequencies of processor cores [7].

The performance of a single-core processor began to reach its physical limits. To keep up with Moore's law and the demand for more computational power, chip designers started to squeeze more cores into a chip and thus began the era of multi-core computing [7]. Initially observed by Gordon Moore, co-founder of Intel, in 1965, Moore's law states that the number of transistors in a chip doubles every two years. Surprisingly, Moore's law describes the industry trends even today [6].

2.1.2 Concurrency Revolution

Historically, the advancement in hardware increased the speed of sequential applications with each new generation of microprocessors. Since the evolution of the single core stalled due to physical limits, such an expectation is no longer valid. Instead,

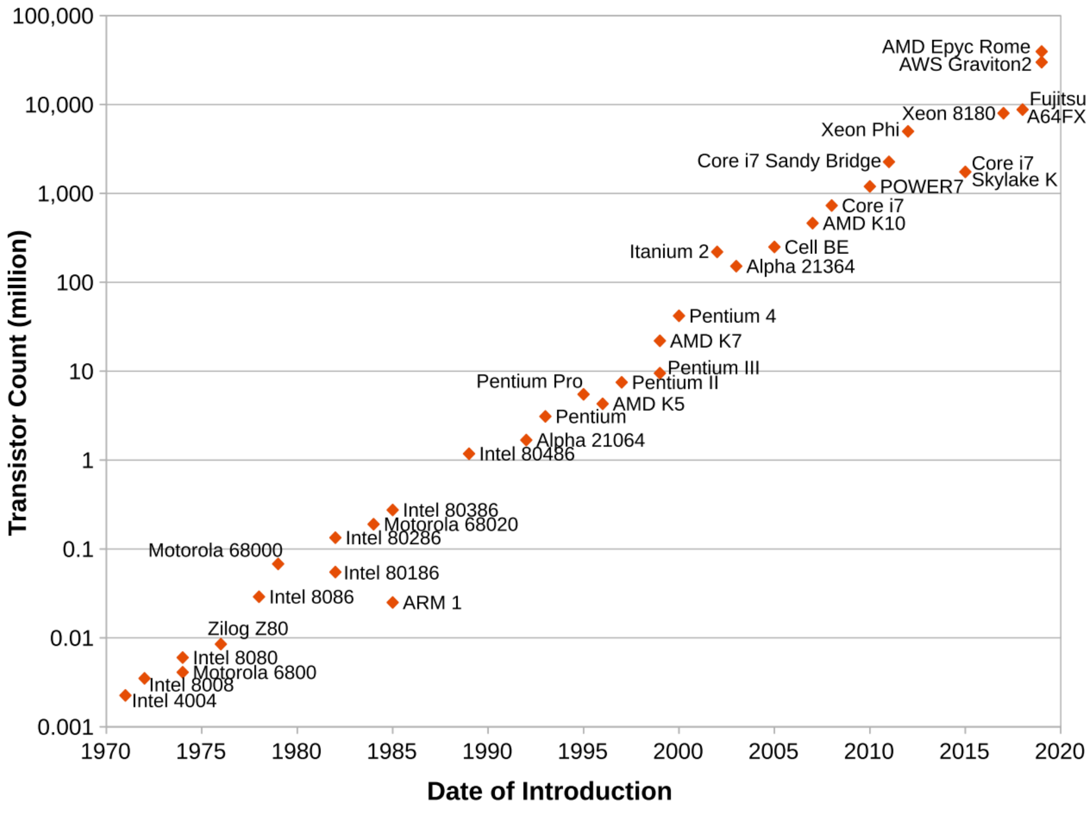


Figure 2.2. Moore's law [7].

the software that will continue to benefit from the significant performance improvement of new processors will be parallel programs with multiple threads of execution, the incentive referred to as the concurrency revolution [8].

A central processing unit is a general-purpose, low-latency-oriented microprocessor. The latest CPU architectures maintain the execution speed of sequential programs while increasing the number of cores, each being a full-featured, highly optimized, sophisticated single-core processor with out-of-order, multiple-instruction issue design supporting hyper-threading [8]. CPUs employ a large on-chip memory cache hierarchy, a few complex arithmetic and logic processing units (ALUs), and complex instruction decoding and branch prediction hardware to avoid stalling, effectively decreasing the latency [7].

In contrast, high execution throughput-oriented many-core GPU architectures contain thousands of simpler cores. GPUs omit sophisticated control logic and memory cache hierarchy and use the saved surface area and power to provide more execution units and memory access channels. The chip is filled with massive amounts of ALUs, floating-point units (FPUs), special function units (SFUs), and memory lanes to dramatically increase the execution throughput and memory bandwidth, hiding the high latency behind the sheer amount of processed data [8].

The maximum peak performance of a processor is given by multiplying the number of cores, the clock frequency, and the number of floating-point operations per clock cycle (FLOPs/cycle). The throughput measurement unit is floating-point operations per second (FLOPS) [6]. The high-end powerful consumer class desktop CPU, Ryzen 9 5950X [21], from the previous generation Zen 3 architecture introduced by AMD in 2022 has 16 cores with a 4.9 GHz clock frequency, capable of executing 32 single-precision

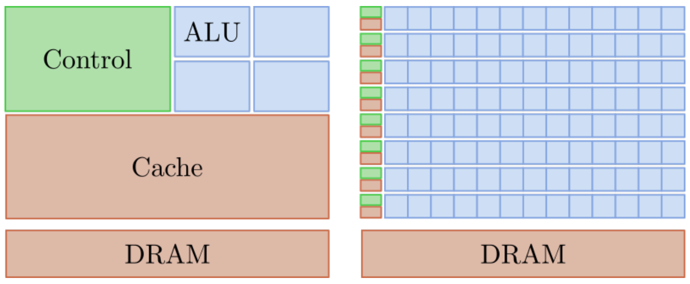


Figure 2.3. Difference between CPU and GPU architecture design. A huge portion of the die area of the CPU on the left is filled with cache hierarchy and control unit to achieve low latencies. The GPU on the right is filled mostly with the computation cores [8].

FLOPs/cycle. The maximum peak throughput is 2,509 single-precision GigaFLOPS. The high-end consumer class GPU, RTX 3090 Ti [27] from the previous generation Ampere architecture introduced by NVIDIA also in 2020, has 10,752 cores operating at 1,860 MHz clock frequency with the execution of 2 single-precision[2] FLOPs/cycle. The maximum peak throughput is whopping 39,997 single-precision GigaFLOPS. Note that even though the 450 watts thermal design power (TDP) of RTX 3090 Ti is way higher than the 105 watts TDP of Ryzen 9 5950X, the GPU offers much better power efficiency of FLOPS per watt.

It is clear now that GPUs are designed as parallel, throughput-oriented computing engines and will not perform well on sequential tasks. Programs with few threads achieve higher performance on low-latency CPUs since many of the cores of GPUs are not utilized [8]. The overhead communication of the GPU with the CPU over a slow PCIe bus creates a data size threshold, below which GPUs are not an effective tool to use. The GPU has to move the data in and out of its main dynamic random access memory (DRAM), introducing high latency in the computation. Having CPU and GPU cores share and access the same memory space promises potentially greater performance [7]. Still, GPUs offer uncontested performance for computing relatively simple, data-parallel tasks on massive amounts of data. The typical scenario and the initial reason for the invention of the co-processor is real-time graphics rendering.

2.2 NVIDIA

Founded in 1993, NVIDIA is a globally renowned technology company specializing in designing and manufacturing advanced graphics processing units. Long established as a global leader in computer graphics, AI, and high-performance computing, its products and solutions have profoundly impacted various industries. NVIDIA GPUs have revolutionized real-time graphics, enabling realistic and immersive gaming experiences and high-quality visual effects in movies and animations. With the introduction of the CUDA parallel computing platform and libraries, their GPUs have become instrumental in accelerating AI and deep learning. The training of the recent GPT-3 language model by OpenAI was conducted on an NVIDIA-powered data center with the performance of 323 ZettaFLOPS, or 10^{12} GigaFLOPS [19].

[2] NVIDIA does not state the single-precision FLOPs/cycle data. Peak single-precision TeraFLOPS is listed instead. FLOPs/cycle is obtained by dividing TeraFLOPS with the number of CUDA cores per GPU and then dividing with GPU clock. The GPU data is in Table 2 of Appendix A in the Ada Lovelace architecture whitepaper [27].

2.2.1 GeForce 256 and 3

Introduced in 1999 and marketed by NVIDIA as the world's first GPU, GeForce 256 graphics card was a specialized acceleration hardware to accelerate real-time graphics. Based on an NV10 graphics chip fabricated with 220-nanometer technology, the GPU offered a performance of 50 GigaFLOPS. After 20 years of technological advancements, RTX 3090 Ti delivers 800 times higher performance with 39,997 GigaFLOPS of performance, keeping up with Moore's law.

The vertex stage of the graphics pipeline became programmable by using shader functions in the first programmable graphics card, NVIDIA GeForce 3, launched in 2001. The GPU included a vertex processing unit for transforming vertices and light shading. Fragment and geometry programmable processing units followed in subsequent GPUs, the first being Radeon 9700 from ATI, later acquired by AMD, introduced in 2002, with fragment shaders defining the behavior of rasterization and geometry shaders generating new vertices. In modern GPUs, light shading is performed per fragment instead of vertex, as in the back days. Graphics programmers express vertex, geometry, and fragment shaders in high-level shading languages like Khronos Group's GLSL or Microsoft's HLSL. The source is compiled into bytecode offline and loaded as a binary by the graphics driver at runtime [4]. Such loading is accomplished through graphics APIs such as OpenGL or Vulkan, which abstracts and exposes the graphics capabilities of the GPU. However, the graphics pipeline with multiple programmable types of processing units with distinct functions has limitations.

The pipeline suffers from extreme variations in workload resulting in an overload of pipeline stages. The number of vertex, geometry, and fragment processing units is fixed, but the computational and bandwidth requirements depend on the behavior of shader functions and properties of the virtual scene. Triangles covering large portions of the screen generate a much higher load on fragment processing units than the vertex units. Conversely, many small triangles result in higher vertex processing demands. To overcome this issue, GPUs employ sophisticated dynamic load-balancing techniques to utilize computation resources as efficiently as possible. However, when a computation of a single stage dominates the execution, the processing units of other pipeline stages will not be guaranteed to be utilized, thus effectively wasting GPU resources [4].

Various data-parallel algorithms have been ported to the GPU using shading languages to exploit its computation power. Using GPU for general-purpose computing such as protein folding, stock options pricing, and SQL queries achieved remarkable performance speed-ups. Nevertheless, the early GPGPU model faced several drawbacks. The programmer was required to know the graphics API and GPU architecture. Problems had to be expressed in vertex coordinates, textures, and shader functions, significantly increasing program complexity. Additionally, random reads and writes to memory were not supported.

2.2.2 Tesla and Fermi

In 2006, NVIDIA launched a GeForce 8800 graphics card running on a G80 chip based on a new Tesla GPU microarchitecture, which introduced unified shader processing units capable of executing any vertex, geometry, and fragment shaders. These units are assigned to process a shader based on the current workload, minimizing the chances of overloading one of the stages of the graphics pipeline [6].

The introduction of unified shaders required a new internal arrangement of the GPU components, a new technology called compute unified device architecture (CUDA). Tesla introduced support for the C programming language, shared memory for concurrent threads, and barrier synchronization for communication between threads, drastically increasing programmability and flexibility. Tesla moved from a single-instruction multiple-data (SIMD) execution model, where a single instruction of a single thread operates on multiple data, to a single-instruction multiple-threads (SIMT). This execution model decodes and executes a single instruction on multiple independent threads, each processing its data on its set of registers, with threads free to branch. GPU became the first many-core massively parallel scalar processor [24].

The successor Fermi architecture presented in 2010 marked another significant leap forward in GPU design. Conceptually unchanged to the present day, Fermi introduced an updated CUDA graphics chip architecture that offered concurrent kernel execution, protection from memory errors when deploying large numbers of GPUs, proper cache hierarchy to enable a vast spectrum of parallel algorithms, faster thread context switching, atomic operations and improved double precision performance. Thanks to these innovations, enormous arrays of GPUs have been massively deployed in data centers, marking 2010 as the leap year for high-performance computing leading to massive breakthroughs and computational achievements of recent years in artificial intelligence and deep learning, medical imaging, physical simulations, and many other workloads.

The new CUDA architecture is based on a streaming multiprocessor (SM) concept containing functional units and programmable CUDA cores processing floating point or integer instructions on individual threads. SMs use a SIMT execution model to execute the warp, a collection of 32 threads. All SMs are mutually independent, thus allowing for splendid scalability. The number of streaming multiprocessors and their functional units depends on the graphics card series and the generation of GPU architecture. The general specifications and features of CUDA-enabled graphics chips are summarized by their CUDA compute capability number (CC) [6].

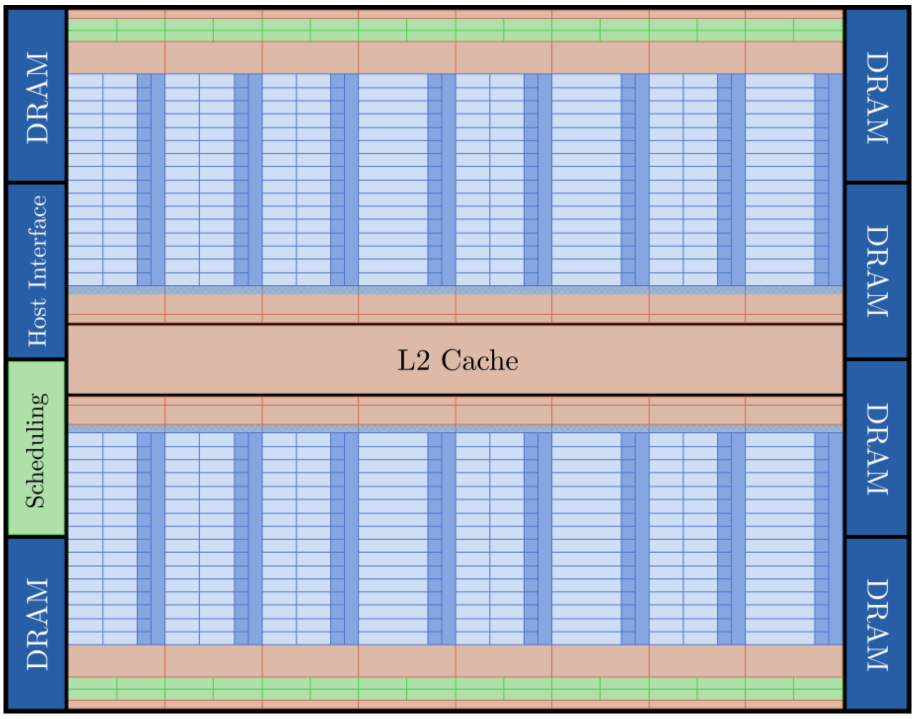


Figure 2.4. Fermi based GPU [24].

Fermi-based graphics chip comprises 16 streaming multiprocessors around a shared level-2 cache. Each SM is a vertical strip whose major part of the surface area is occupied with execution units, with the rest left for thread schedulers and on-chip memory. SMs are connected through high-capacity memory lanes to off-chip DRAM. Thread blocks are scheduled to various SMs by global work distribution, and each warp scheduler in SM distributes warps of 32 threads to its execution units. Since the Fermi architecture, different kernels can execute concurrently, allowing maximum utilization of GPU resources [24].

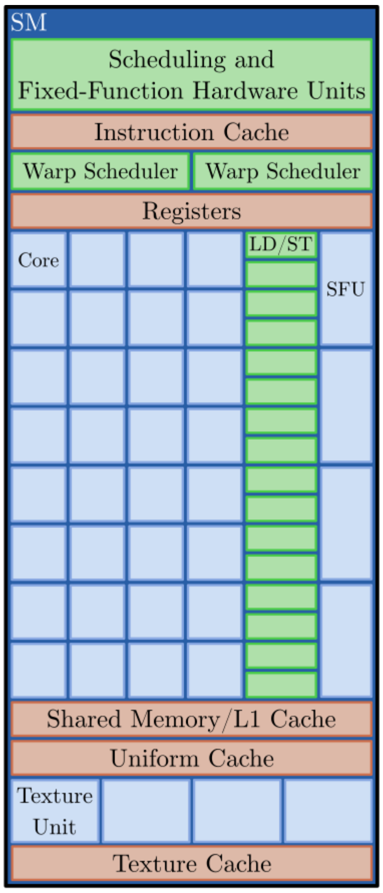


Figure 2.5. Streaming multiprocessor of the Fermi based GPU [24].

Each SM contains 32 CUDA processors. Each CUDA core has a fully pipelined ALU and FPU unit. 16 load and store units calculate the source and destination addresses for 16 threads per clock, loading and storing the data in the on-chip cache or off-chip DRAM. Four SFUs execute transcendental instructions such as sine, cosine, and square root. Each SFU executes one instruction per thread per clock. Each SM features two warp schedulers and two instruction dispatch units, allowing two warps to be issued and executed concurrently. On-chip shared memory enables threads within the same thread block to cooperate, facilitating extensive reuse of on-chip data, and the level-1 cache reduces slow off-chip DRAM traffic [24].

The new NVIDIA GPU instruction set (PTX) provides a stable architecture spanning multiple GPU generations, including the most modern ones. The programming model of the PTX set scales across GPU sizes, providing code redistribution. Unified address space between thread private local, block shared, and global memory enables the proper use of pointers and thus fully supports C++ [24].

Since introducing scalable Tesla and Fermi CUDA-based microarchitectures, NVIDIA launched new product lines next to its GeForce GTX gaming series. The Quadro series with more connectors targeted professional workstations, accelerating CAD software, graphics production, machine learning, and other calculations before being discontinued and fused with GeForce GTX into the GeForce RTX series. The Tesla series targets GPGPU by increasing the number of CUDA cores and memory capacity at the expense of fixed-function graphics units such as raster operation units (ROPs) [6]. The Tesla series was rebranded as NVIDIA Data Center GPUs. Later, the Tegra series targeted embedded systems and mobile devices, and the Jetson series provided AI computation power to robots and drones. The variety of products underlines the flexibility and scalability of CUDA.

2.2.3 Volta and Turing

Since the introduction of Fermi architecture, the traditional graphics pipeline could be computed parallelly in a streaming fashion. Pixels could be fed to ROPs and written to the frame buffer, the depth buffer could reject fragments, and new vertices could be fetched, all done simultaneously. Even though each triangle has to go through the same logical steps, multiple triangles in different stages can be processed in parallel. The fixed pipeline present since the first GPUs became decoupled from the underlying hardware and became purely logical [12].

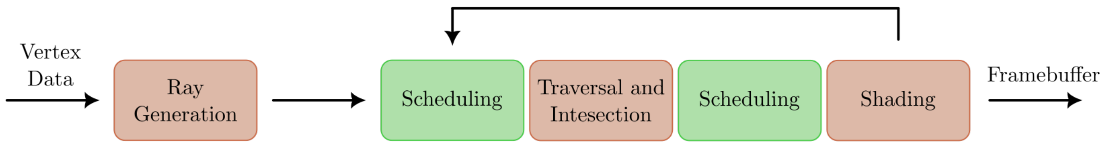


Figure 2.6. Ray-tracing pipeline [26].

The rendering pipeline can now be implemented in software with GPGPU techniques or even changed entirely. Such is the case with the ray-tracing pipeline, a rendering process different from traditional rasterization. Ray-tracing was used to render images in non-real-time applications, such as offline rendering of movie frames, and was enabled to run on GPUs through the software OptiX CUDA library [28]. However, both rasterization and ray-tracing pipelines perform concrete computations that could be accelerated through the hardware, and the software freedom of programmability would not provide any additional value. Rasterization has been accelerated with hardware ROPs and texturing with texture units since the introduction of unified CUDA cores in Tesla architecture.

Ray-tracing has been hardware accelerated by introducing special-purpose fixed-function ray-tracing cores in Turing architecture. The TU102 gaming chip was the first GPU with 72 ray-tracing cores, each including two specialized units, one for bounding box tests and the second for ray-triangle intersection tests. When tracing a ray, the algorithm performs bounding volume hierarchy (BVH) traversal until hitting a triangle and its color at the ray intersection point. Software emulations required thousands of instructions per ray, making the process impossible on GPUs in real-time without hardware acceleration. In the TU102 chip, streaming multiprocessors offload BVH traversal to RT cores returning hit or no hit results, while SMs are freed up to do other work. Turing architecture manages to process more than 10 billion rays per second. In practice, a hybrid pipeline is used, with ray-tracing rendering physically accurate shadows, reflections, and refractions, while rasterization renders everything else [26].

Typical GPGPU use cases, such as machine learning, have also been hardware accelerated. Before the Turing architecture, the data center Tesla V100 chip based on

Volta architecture was the first GPU equipped with tensor cores, special-purpose acceleration units performing matrix multiplications. The chip included 640 half-precision floating-point (FP16) matrix multiplication hardware units and 5,120 CUDA cores. While the peak single-precision performance of all CUDA cores combined reached 15.7 TeraFLOPS, tensor cores reached 125 half-precision TeraFLOPS, a 12 times speed-up compared to the CUDA-based software implementation of matrix multiplication in the previous Pascal architecture. With the V100 chip, NVIDIA introduced NVLink system interface technology to interconnect up to eight V100 accelerators at up to 300 GB/s on a single server, outpacing the traditional 32 GB/s PCI Express interface used in consumer personal computers. The improved memory system combined with tensor cores led to a massive performance increase in data centers [25].

Chapter 3
General-Purpose Computing

Programming GPUs requires a different paradigm compared to CPUs. In order to put all the computational power to use, all the cores must be put under the workload, with even more threads needed to hide the long latencies of DRAM memory accesses. Enough data has to be processed since the GPU and CPU memories are disjoint, and the data transfer overhead between the two will inevitably slow down the system for not big enough workloads. Additionally, due to the inner architecture of graphics chips, the computational tasks suitable for GPUs must run independently in parallel. With many inherently sequential tasks, CPUs and GPUs must coexist to build an effective working system. The workloads that benefit from running on GPUs the most are the data-parallel workloads such as scientific computing, fluids dynamics simulations, weather forecasting, business analytics, machine learning, and real-time rendering.

Many GPGPU development platforms can solve the task upon selecting a suitable data-parallel workload. On one side of the spectrum are tools that require explicit problem decomposition and offer low-level access to the GPU, like CUDA [17] and OpenCL [18], and on the other side, tools like OpenACC [33] let the compiler do all the work through the use of compiler directives, such as data migration and thread spawning. Some standards, like OpenMP [34], developed initially for shared-memory parallel CPU programming, now also support targeting GPUs. Finally, many higher-level libraries written in low-level tools exist, such as the CUDA-based Thrust [32] library utilizing a set of container classes and algorithms to automatically map computations to GPU threads.

3.1 CUDA

The Compute Unified Device Architecture is the hardware and software platform enabling NVIDIA GPUs to execute programs written in an interface extension to languages like C, C++, or other parallel development tools like Thrust. The CUDA compiler uses programming abstractions to leverage parallelism built into the CUDA hardware, lowering the burden of programming. NVIDIA offers the whole development ecosystem besides third-party toolchains utilizing CUDA like OpenCL or OpenACC. There are plenty of libraries collectively called CUDA-X [32] with support for accelerated linear algebra, Fourier transforms, random number generation, parallel algorithms, computational lithography, image and video encoding, signal processing, communication, deep learning, and others. Tools like profiling and debugging tools, GPU cluster management, monitoring, containerization, scheduling, and orchestration are available [11]. The following text describes programming in CUDA runtime API, a syntactic extension of C/C++.

3.1.1 Programming Model

A typical execution of a CUDA program consists of copying the input data and the compiled program from the CPU to the GPU. The program is loaded and executed, and the results are copied back. When typing code, the programmer may mark a function to run on the GPU, called the device, with a `__global__` keyword. Such a function is called a kernel, which is instantiated across multiple threads that use this function to process data on the device during the computation. The kernel, which must be declared with the `void` return type, is executed on the CPU side, referred to as the host, and the programmer chooses how many threads should run the kernel and their organization into blocks and the blocks into a grid. Such execution configuration is denoted with `<<< >>>` syntax. Kernels are executed asynchronously. The host does not wait until the computation on the device completes unless the explicit `cudaDeviceSynchronize` barrier statement is used [7].

```
// Kernel function executing on the device (GPU)
__global__ void kernel() {
    ...
}

// Function executing on the host (CPU)
int main() {
    // A grid of 40x20 thread blocks
    dim3 blocks(40, 20);

    // The kernel is launched with 32 threads per each block
    kernel <<< blocks, 32 >>> ();

    // Wait for the kernel to finish
    cudaDeviceSynchronize();
}
```

Thread blocks and the grid can be multi-dimensional. Each thread is aware of its position in the grid and block hierarchy with built-in three-dimensional structures `blockDim`, `gridDim`, `threadIdx`, and `blockIdx`. Threads can synchronize within a block through `__syncthreads` barriers. Blocks are mutually independent and can thus be executed in arbitrary order, allowing for splendid application scaling on any number of streaming multiprocessors. Kernels can only operate on the data located on the device memory, which must be copied to the device prior to the execution of the kernel [7].

```
__global__ void kernel() {
    // Id of a thread executing an instance of the kernel
    unsigned int id = blockIdx.x * blockDim.x + threadIdx.x;

    // The execution of all threads stops here
    // until all threads reach this barrier
    __syncthreads();
}

int main() {
    int host[] = { 1, 2, 3, 4, 5 };
    int *device;
    // Allocate memory on the device
    cudaMalloc(&device, 5 * sizeof(int));

    // Copy memory to the device
    cudaMemcpy(device, host, 5 * sizeof(int), cudaMemcpyHostToDevice);

    // Free the allocated memory
    cudaFree(device);
}
```

The kernel can branch and call other functions, which must be specified in the code with the `__device__` keyword, or even call other kernels through dynamic parallelism. CUDA API functions return error flags structure `cudaError_t` that indicates whether an error has occurred. It is possible to check for a potential error of a kernel call with a `cudaPeekAtLastError`. The API supports many atomic operations above the data in the global or shared memory [8].

```
// This function can only be called from a kernel
__device__ void device_func() {
    ...
}

__global__ void kernel() {
    ...
}

int main() {
    // Error handling of CUDA functions
    int *pointer = 0;
    if (cudaFree(pointer) != cudaSuccess) {
        ...
    }

    // Error handling of kernels
    kernel <<< 10, 32 >>> ();
    cudaError_t error = cudaPeekAtLastError();
    if (error != cudaSuccess) {
        ...
    }
}
```

3.1.2 Memory Model

Threads can access various types of memory on the graphics card, which varies in speed, capacity, and visibility to the programmer. Each thread can access a vast but slow off-chip global DRAM memory. Accessing it lasts hundreds of clock cycles, and the device usually hides global memory read and writes by switching the execution context to another thread. Part of the global memory, called constant and texture memory, is read-only. Accesses to the off-chip memory are handled through the cache hierarchy on the chip. The global memory, including the constant and the texture part, is the only type of memory that the host can access. The programmer can use `cudaMalloc`, `cudaMemset`, or `cudaFree` functions [6].

Local scalar variables declared in a thread are usually stored in registers located in each streaming multiprocessor, and accessing them lasts a single clock. Additional local variables are stored in local memory when the register space runs out. However, the local memory is part of the off-chip DRAM, and the reads and writes are slow. Shared memory, denoted with the `__shared__` keyword inside a kernel, is used to exchange data between threads inside a block. Since it is located on the chip, shared memory is used by the programmer to store frequently accessed data to avoid global memory accesses. The shared memory shares the same physical memory block with the level-1 cache of each SM. Using less shared memory translates to a larger cache [8].

Type	Location	Cache	Access	Visibility
Register	on-chip	—	read/write	1 thread
Local	off-chip	yes	read/write	1 thread
Shared	on-chip	—	read/write	all threads in a block
Global	off-chip	yes	read/write	all threads and the host
Constant	off-chip	yes	read	all threads and the host
Texture	off-chip	yes	read	all threads and the host

Table 3.1. Specification of CUDA memory types [6].

3.1.3 Execution Model

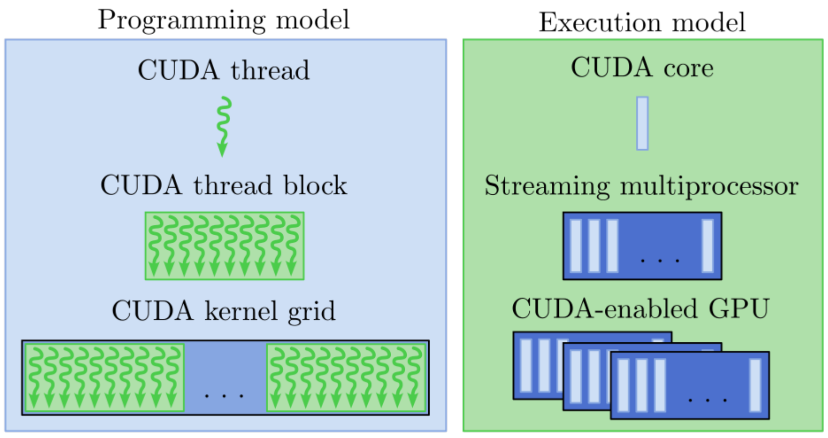


Figure 3.1. Correlation of the CUDA programming and execution model [11].

The execution of the kernel is mandated by the global scheduling unit, which can handle more kernels concurrently. The kernel is usually split into thread blocks, which are dynamically load balanced and assigned to streaming multiprocessors. Each thread block is executed on a single SM. Multiple blocks can be mapped to the same SM, with those currently not executed marked as resident [6].

Each block of threads constitutes multiple warps, groups of 32 threads running in parallel at once at any given time. Each thread in the warp executes the same instruction but on different data in the SIMT architecture fashion. The warp scheduler of each SM plans the execution of threads of assigned blocks by splitting them into warps, marking the ones currently not executed as resident. Each thread from an active warp is assigned a CUDA core from the SM and executes an instruction. The whole warp is then context-switched for another resident warp. The process repeats until all the resident warps of all the resident blocks are processed. The warp scheduling is hardware accelerated and thus has no overhead [6].

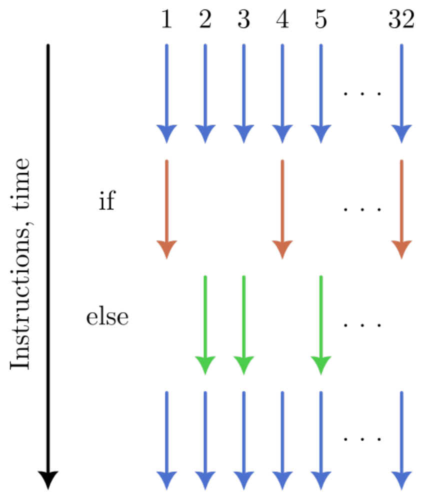


Figure 3.2. Warp divergence on conditional jumps [13].

SIMT hardware parallelism has the drawback of warp divergence in conditional statements, where some threads of the warp may execute other instructions from the others. When a conditional statement is handled, those threads of the warp that pass the condition are marked as active, while the others are masked out. All the threads execute the same instruction, but only those marked as active write results to the memory. Upon finishing the branch, the program execution returns to the conditional statement and repeats the masking of the threads for the other branch. In the worst case, each warp thread may execute its branch sequentially, repeating the same code segment 32 times [6]. To overcome the issue of warp divergence, conditional jumps should be avoided as much as possible.

3.1.4 Compilation

The `nvcc` compiler driver tool must be used to compile the CUDA program present in `.cu`-suffixed files. The host code is compiled into object files, serving as the input to the linking phase. The device code is compiled into a `.ptx` file, a portable assembly format, or a `.fatbin` binary file encapsulating multiple PTX and `.cubin` binary files that target a specific GPU. NVIDIA does not preserve binary compatibility across GPU architectures, allowing them to innovate and produce radical new designs. The PTX intermediate code is compiled to the `.cubin` format with a just-in-time compiler (JIT) before the execution, adding a little startup overhead during the first invocation of a kernel. CUDA can embed both `.cubin` and PTX versions of the device code in the produced executable. The programmer can control the virtual PTX architecture and actual device architecture through `nvcc` compiler flags [7].

Chapter 4
Graphics Pipeline

A graphics system generates images representing a virtual camera view of a virtual scene defined by the geometry, orientation, and material of objects and characteristics of light sources. Such rendering computation is performed on a graphics pipeline, a series of stages. Data flows between these stages in streams of fundamental graphics entities called primitives, on which each stage performs a specified operation. It is a process of displaying 3D vector graphics on a 2D raster screen.

4.1 Logical Pipeline

Back in the day, before the release of G80's unified architecture, the actual graphics pipeline logic was sealed into the GPU's hardware. The pipeline was represented directly on the chip as the series of computational stages. G80 GPU introduced a unified computing unit that could execute vertex and fragment shaders and dynamically balance the load between them. However, the process of primitive assembly, rasterization, and other stages was still serial. With the introduction of Fermi architecture, the pipeline became fully parallel. Once rooted in the chip, the graphics pipeline is now a purely logical, fully concurrent series of steps executed by many unified computing engines and some fixed-function hardware acceleration units in parallel [13].

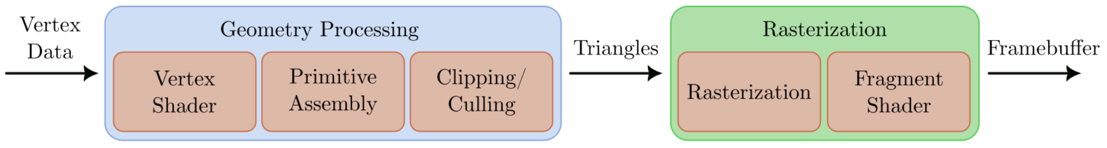


Figure 4.1. Graphics pipeline [3].

The logical pipeline comprises a geometry processing section with input 3D geometry projected to the 2D plane and a rasterization section that colors the pixels covered by the geometry by calculating its screen coverage and lighting. To initiate rendering, the vertex processing stage constructs a stream of vertices from the input data, transforms them, and projects them from 3D coordinates into a 2D screen space. The primitive processing stage uses vertex topology to group vertices into a set of primitives, most commonly triangles, and performs culling and clipping operations. The fragment processing stage samples each primitive densely in screen space into fragments containing various data, such as a distance from the virtual camera or a surface color. The last pixel stage calculates the final color and the fragment's contribution to the corresponding pixel in the frame buffer [4]. All the processing happens in a single draw call, many times a second.

4.1.1 Life of a Triangle

Before a draw call, the application must set an ordered list of vertex data to send to the pipeline. Such data usually includes vertex position, normals, and texture coordinates. The data is grouped through indexing into primitives, basic drawing shapes like triangles, lines, and points that will be processed. The vertex processing stage is almost fully programmable [12].

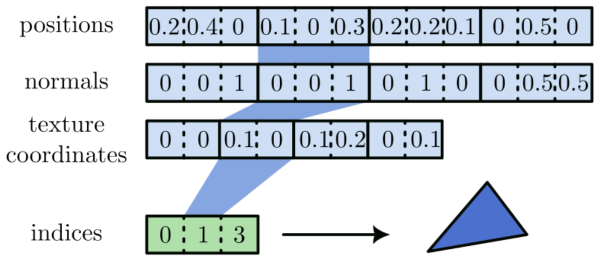


Figure 4.2. Assembling a triangle by indexing vertex data buffers.

The vertex shader processes each input vertex, outputting it to the following stages based on an arbitrary programmer-defined program called shader. Generally, vertex shaders are expected to transform vertices from the initial position to clip-space coordinates. Such operations are computed as matrix multiplications of transformation matrices. While the vertex shader pipeline stage is not optional, the tesselation stage is. Input primitives can be tessellated, or tiled, into many smaller connected primitives by a tessellation shader, which is aided by a fixed-function tessellation accelerator. Another optional vertex stage is a geometry shader, which processes each incoming primitive, returning zero or more output primitives [12].

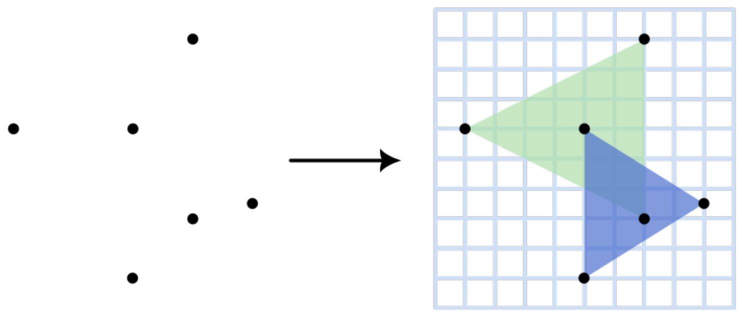


Figure 4.3. Primitive assembly [4].

After the shader-based programmable vertex processing stage, vertices undergo several fixed-function steps. The primitive assembly groups output vertices into a sequence of primitives, and if tessellation or geometry shaders are active, a limited form of the assembly is executed before these stages. The formed primitives outside the viewing volume are discarded, or culled. In contrast, the ones on the boundary are split into several primitives inside the volume through the process called clipping. The vertex positions are then transformed from clip space to screen space. Back-face culling usually happens in this stage, with triangles facing from the camera being discarded [12].

The rasterization stage processes each triangle, determining its screen coverage and outputting a sequence of fragments representing a pixel candidate in the output frame

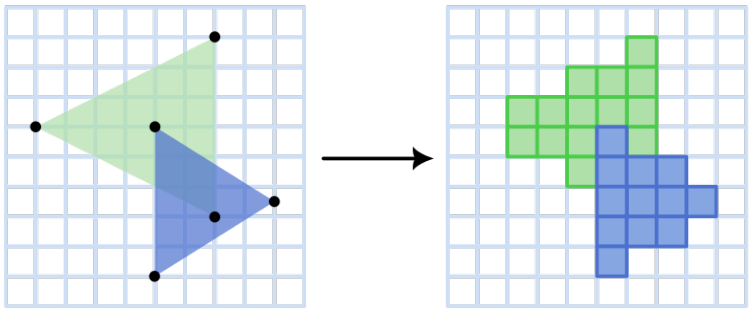


Figure 4.4. Rasterization [4].

buffer. Fragment holds its position, the depth from the camera, and arbitrary data interpolated from the values of primitive vertices. Rasterization is entirely and efficiently hardware-accelerated on the chip.

Fragments are processed in a fragment or pixel shader. The shader's output is a color and depth, and stencil value. These are used in the final visibility tests of the raster operations stage. The scissor test discards fragment data outside of a specified rectangle of the screen, the stencil test fails if the provided stencil value does not compare against the programmer-provided value, and the depth test discards fragments occluded by another fragment closer to the camera. Finally, the color blending of various fragments happens, usually in the case of transparent objects [12]. The result of such a process is a rendered image of the virtual scene on the screen.

4.2 Hardware Pipeline

The hardware design of the pipeline on the GPU chip is a wonderfully complex and meticulously crafted engineering achievement, guaranteeing that all kinds of inputs are processed efficiently [2]. GPUs cope well with variable workloads, typical for real-time graphics rendering. Since the introduction of Fermi, NVIDIA GPUs have followed a similar principle architecture. The global scheduler manages the entire graphics workload between multiple graphics processing clusters (GPCs) with multiple SMs and a raster hardware acceleration unit. Many interconnecting memory lanes on the GPU allow work migration across GPCs and additional fixed-function accelerators like render output units (ROPs) [13].

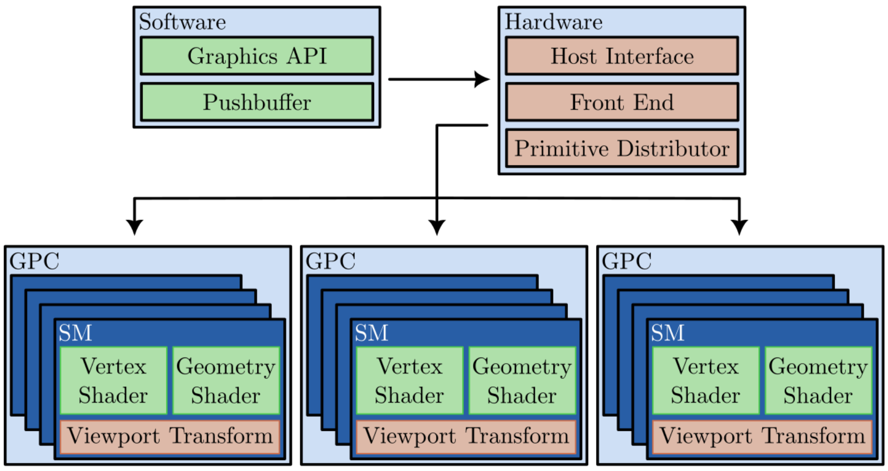


Figure 4.5. Geometry processing [13].

Rendering may begin once a vertex buffer is passed to the DRAM of the GPU. The program makes a draw call in a graphics API. The GPU driver validates the input configuration and inserts the command in a GPU-readable encoding inside a push buffer. When enough commands are accumulated in the buffer, its contents are sent through the PCIe host interface and processed by GPU's front end. On some game consoles, the CPU and GPU share memory space with a unified memory architecture and thus avoid the delay of feeding the data to the GPU, but that is not the case on personal computers [14]. The work commences in the primitive distributor by processing indices and generating triangles fed to GPCs [13].

The scheduling engine of one of the SMs within a GPC fetches the vertex data, which are then executed in batches of warps of 32 threads according to the CUDA execution model. The threads are regularly context-switched to hide the long latencies of memory access. Once the warp completes all instructions of vertex shaders, results are processed by the viewport transform, which performs clipping and feds the data to rasterizers. All the communication between stages is utilized through level-1 and level-2 caches [13].

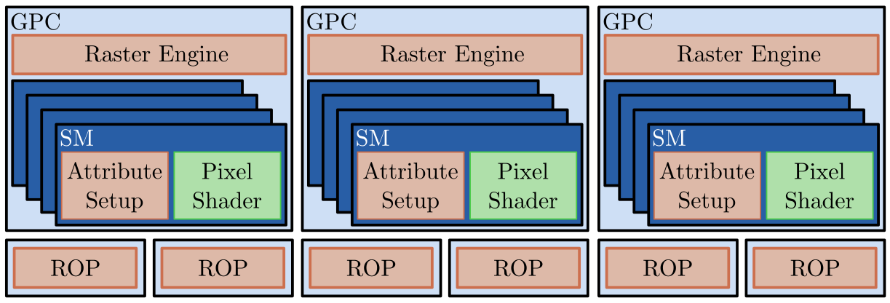


Figure 4.6. Rasterization [13].

The screen is split into multiple tiles, with each raster engine covering a portion. The triangle's bounding box determines which engines should process the triangle, which may leave the GPC in which it was processed until now and be assigned to one or more GPCs through a work distribution crossbar. The attribute setup converts the triangle data to pixel shader format, and the hardware-accelerated raster engine can begin its rasterization and back-face culling. Again, the pixel shader is executed in batches of warps according to the CUDA execution model. Thirty-two threads process eight 2×2 pixel regions, calculating colors and depth values at each pixel. At this point, the original API ordering of triangles must be preserved before passing the data to one of the ROPs, which performs depth-testing and blending with the frame buffer through stream outputs [13].

Some stages of the hardware graphics pipeline were omitted in this description to keep it reasonably straightforward and tightly coupled with the thesis pipeline implementation presented in the following chapter. Another integral part of the GPU pipeline is texture sampling with its hardware acceleration through texture memory, texture cache hierarchy, and texturing units. It performs texturing of triangles, a complicated process of texture fetching, filtering, and interpolation. Another vital fixed-function piece of hardware is a tessellation unit used to accelerate tessellation shaders. Finally, graphics APIs now allow running compute shaders that execute the same way as GPGPU programs described in the CUDA programming model [14].

The actual hardware implementation of the pipeline uses a similar screen tiling as the software CUDAraster [2]. However, the tiling is performed per triangle thanks to the GPU's atomicity of the hardware depth buffer instead of globally per screen. Load balancing and stage parallelization, which is natural to any CUDA computation workload, offers the same benefits as the similar one in the cuRE software pipeline [3]. As such, it is virtually impossible to reach the performance of the hardware-accelerated pipeline with GPGPU techniques, and the only benefit of software implementations lies in the extended programmability and flexibility.

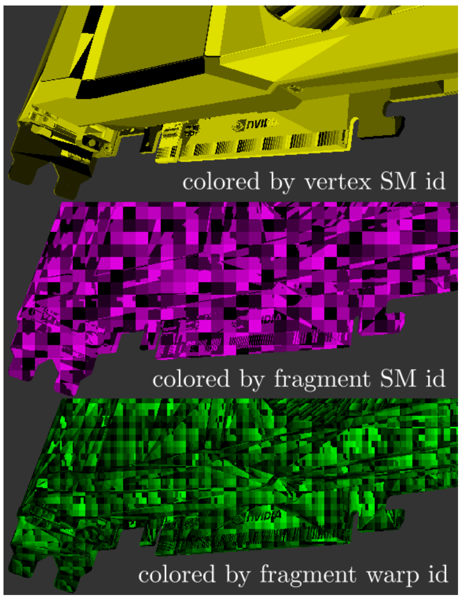


Figure 4.7. Tiling of the screen of the hardware rasterization [13].

4.3 Future of Real-Time Rendering

Although an incremental change toward a more flexible hardware pipeline is difficult due to the complexity of the data flow and reliance on fixed-function hardware units, the evolution of the GPGPU techniques and the subsequent introduction of new hardware acceleration units enabled slow but steady advancement of the graphics pipeline.

4.3.1 Global Illumination

The traditional real-time rendering employes local illumination, with each object in the virtual scene shaded individually. Such lighting computation is fast, easily implemented on the hardware, and produces sufficiently life-like results. In the real world, however, the reflected light from an object interacts with other objects in the scene, producing additional light sources, shadows, reflections on glossy surfaces, and caustics caused by transparent objects. The light reflection can significantly alter the appearance of environments not directly lighted by a light source. The results of global illumination methods are almost indistinguishable from real photographs [9].

Direct or local illumination is a lighting simulation of a light ray visible after a single bounce from an object, with the Phong illumination model [9] being the typical example. One of the classical methods of computing global illumination is the method of ray tracing or a similar method of path tracing. A ray from each pixel of the camera view is emitted into the scene, each time with a slight direction variation, bouncing off the objects, with each bounce contributing to the pixel color. Naturally, more ray bounces and traced rays produce a better visual result. However, tracing rays and calculating

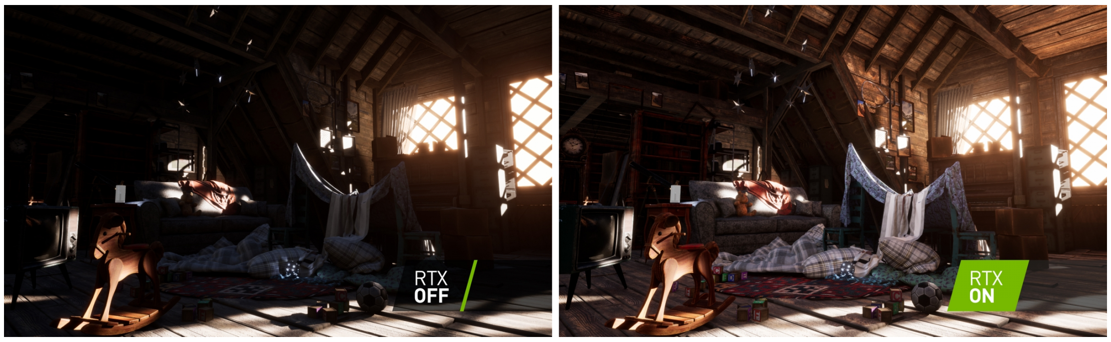


Figure 4.8. Global illumination computed through NVIDIA's hardware-accelerated ray-tracing RTX technology [29].

intersections with triangles is a computationally demanding task. A commonly used data structure for ray tracing is a bounding volume hierarchy (BVH) tree. The scene is split into a hierarchy of bounding boxes, from the scene objects to the individual triangles. The ray tracing algorithm traverses the BVH structure, comparing the traced ray with bounding boxes until reaching a triangle it intersects [9]. BVH traversal was recently accelerated by introducing hardware ray-tracing cores in NVIDIA GPUs, enabling ray tracing in real-time, a domain used until now primarily for offline renderings, such as in the movie industry [28].

4.3.2 Artificial Intelligence

The path tracing algorithm introduces noise in the output image, requesting a higher number of rays sent from each pixel, thus increasing the computational demands to overcome this issue. The introduction of tensor cores, the hardware units for matrix multiplications, accelerated many artificial intelligence workloads. NVIDIA's AI-accelerated denoiser was trained on thousands of images to smooth the noised output, with the neural network available to developers to increase the quality of results even further [30].

NVIDIA deep learning super sampling (DLSS) technology utilizes GPU AI performance to generate new frames and displays higher than rendered resolution through image reconstruction. A neural network is trained on NVIDIA supercomputers on targeted video games to run the game at a lower resolution for increased performance or higher graphical settings, with DLSS upscaling the image. In addition to running on tensor cores, the deep learning model can also run on the CUDA shader cores. The second version of DLSS introduced a frame-generation algorithm to double the framerate, with the third version introducing deep learning dynamic super resolution (DLDSR), an AI-assisted technology of down-scaling an image rendered at a higher resolution to increase the visual quality [31].

Chapter 5
Implementation

The thesis implements a rendering pipeline written using GPGPU techniques in C and its CUDA extension without using any graphics API. It only runs on CUDA-enabled GPUs and Windows systems. Apart from the CUDA runtime API and Win32 Windows API [37] for creating and managing a window, the implementation does not rely on third-party libraries, and everything is written from scratch. The design of the rasterization stage and screen splitting into bins and tiles conceptually follows CUDAraster's [2]. The reader is advised to read the user manual with input controls, the compilation process, and the source code structure in the thesis appendices.

5.1 Pipeline Design

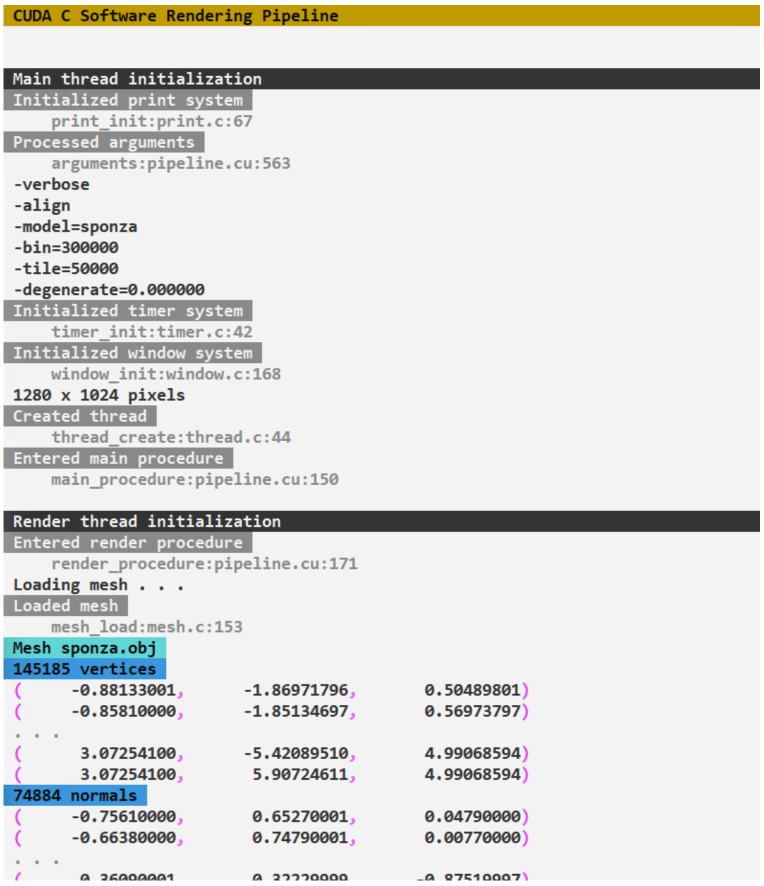
The renderer focuses on effective rasterization and a basic lighting illumination model. The implementation renders a scene with a single mesh, a single point light, and a camera orbiting around the origin. The user can control the behavior with keyboard inputs. There are a few constraints regarding the design. Namely, the pipeline only renders triangle primitives shaded by the illumination model. It does not perform any texturing, blending of the pixels with the framebuffer, and thus rendering of transparent objects. It does not retain primitive order imposed by current modern graphics APIs and does not perform proper near and far plane clipping. Lastly, the window of the application cannot be resized.

The pipeline consists of scene fetch, vertex shader, primitive assembly, rasterization, pixel shader, and raster operation stages. The scene fetch stage fetches scene configuration data such as light position and builds transformation matrices passed to the GPU. The vertex shader stage executes two kernels, one transforming vertex positions to clip space and generating additional vectors for computing per-fragment shading later in the pipeline, with the second kernel transforming vertex normals. The primitive assembly stage launches a single kernel that groups a stream of vertices and normals to triangle primitives using an index buffer. It performs a back-face, view frustum, and degenerate culling, transforming clip space coordinates to the screen space. The rasterization stage launches three kernels, one sorting triangles from the screen to bins, the second sorting triangles from each bin to tiles, and the third performing the actual rasterization per each tile pixel. The rasterizer computes the barycentric coordinates [42] of each pixel in the processed triangle from a tile queue, interpolates the data from its three vertices, and performs a per-pixel depth test that does not require a global screen-wide depth buffer. The resulting fragments, with a single fragment per pixel, are processed in the pixel shader stage. Its single kernel performs pixel coloring by utilizing the Blinn-Phong model [9], a computationally more efficient version of the Phong illumination model [10]. Finally, the raster operation stage copies the GPU frame buffer to the screen buffer.

5.2 Application Systems

The draw call happens periodically multiple times a second, targeting 60 frames per second. The rendering logic is performed by a render thread decoupled from other application logic handled by the main thread. The renderer accepts input and updates the scene even when the draw call takes hundreds of milliseconds to process under heavy rendering workloads. Mutexes guard the exchange of the scene data between the threads. The evaluation of the renderer is possible thanks to a verbose output to a terminal and a timer system with a microsecond resolution. The math implementation used in the pipeline is based on the cglm library [39], a C rewrite of C++ OpenGL mathematics [40]. The custom implementation is provided since the thesis pipeline uses the z-axis as the vertical axis, while the y-axis is generally used instead. The renderer is memory hungry since it copies all the necessary data to the GPU's DRAM at the initialization and leaves it there for the rest of the application execution.

5.2.1 Initialization



```
CUDA C Software Rendering Pipeline

Main thread initialization
Initialized print system
    print_init:print.c:67
Processed arguments
    arguments:pipeline.cu:563
-verbose
-align
-model=sponza
-bin=300000
-tile=50000
-degenerate=0.000000
Initialized timer system
    timer_init:timer.c:42
Initialized window system
    window_init:window.c:168
1280 x 1024 pixels
Created thread
    thread_create:thread.c:44
Entered main procedure
    main_procedure:pipeline.cu:150

Render thread initialization
Entered render procedure
    render_procedure:pipeline.cu:171
Loading mesh . . .
Loaded mesh
    mesh_load:mesh.c:153
Mesh sponza.obj
145185 vertices
(     -0.88133001,     -1.86971796,       0.50489801)
(     -0.85810000,     -1.85134697,       0.56973797)
. . .
(      3.07254100,     -5.42089510,       4.99068594)
(      3.07254100,      5.90724611,       4.99068594)
74884 normals
(     -0.75610000,      0.65270001,       0.04790000)
(     -0.66380000,      0.74790001,       0.00770000)
. . .
(      0.36090001,      0.32229999,      -0.87519997)
```

Figure 5.1. Verbose output of the print system after launching the debug build of the pipeline.

The application starts by initializing the print system, which creates a console window for various informative text outputs. Many macros are provided in the code for various levels of information, including a trace output. Every section of the code is adequately checked for errors, correctly aborting the application execution and informing the user if anything fails. The command line arguments are processed, returning the default values if none are provided. Next, the timer system is initialized, enabling microsecond resolution for timing and millisecond resolution for thread sleep. The timers evaluate the render performance and the correct frame rate. Then, the window system is initialized. The application disables DPI scaling, so a pixel from the frame buffer stays a pixel on the screen. A created window with fixed resolution sets up a frame buffer by creating a Win32 API bitmap that fills the window frame. Finally, the render thread is created, and the main thread enters a loop, where it awaits and processes window events.

The render thread begins its execution by creating timers used to evaluate the execution of each stage and to keep the targeted 60 seconds frame rate. Each stage is implemented in a separate source file and needs to be initialized with pointers to the data from other stages that flows through the pipeline. Each stage module then holds multiple local static pointers, pointing to data structures in other stages. Additionally, since much of the data of the pipeline stages resides in the GPU memory, the buffers that hold this data are allocated on the device in advance before the first draw call. The buffers are not zeroed out since all the data is overwritten with each pipeline render pass. The initialization of stages does not follow the actual stage execution order of the pipeline since some buffers have to be allocated prior to others.

```
/* render_procedure */

// Timer
timer_t timer = { 0 };

scene_init(
    pixel_light_shininess,   // Pointer to pixel shader shininess
    vertex_light_position,   // Pointer to vertex shader light position
    vertex_view,             // Pointer to vertex view matrix
    vertex_view_model,       // Pointer to vertex view model matrix
    vertex_inverse_tranpose, // Pointer to inverse transpose vm matrix
    vertex_pvm);             // Pointer to pvm matrix

vertex_init(
    size_vertices, // Number of vertices
    vertices,      // Vertex buffer
    size_normals,  // Number of normals
    normals);      // Normal buffer

primitive_init(
    vertex_vertices, // Pointer to vertex shader vertices
    vertex_normals,  // Pointer to vertex shader normals
    size_indices,    // Number of triangles
    indices);        // Index buffer

rop_init();

pixel_init(
    rop_framebuffer, // Pointer to raster operation framebuffer
    light_constant,  // Light constants used as weights of components
    light_ambient,   // Pointer to ambient light component
    light_diffuse,   // Pointer to diffuse light component
    light_specular); // Pointer to specular light component

rasterize_init(
    primitive_size_triangles, // Number of triangles
    primitive_triangles,      // Primitive assembly triangle buffer
    pixel_fragments);         // Pointer to pixel shader fragment buffer
```

After setting up timers, the scene is initialized. First, it stores pointers to various pixel and vertex shader variables, such as the light position and the transformation matrices. Then a single mesh is loaded with a Wavefront object [41] format loader written from scratch, which only supports loading vertex positions, normals, and triangles. The included convert utility program or modeling software, such as Blender [35], can convert any model to the suitable .obj format supported by the application. The user is advised to consult the user manual in the appendix to achieve this. After loading the object, the camera and the light are set to default values. The entire scene configuration is mutex guarded.

```
/* scene_init */

// Mutex to guard against data races of the main and render threads
mutex_create();

// Load mesh
load_mesh();

// Set up the rest of the scene by assigning default values
...
```

The vertex shader is initialized with the loaded mesh data. The stage allocates memory on the GPU for constant position and normal buffers, which hold the reference vertex positions and normals. The data of these buffers does not change throughout the entire application execution. Additional two buffers for transformed vertex positions and normals are allocated. Since the data structure used for vectors and matrices is essentially a type-defined array residing in the CPU's RAM, it cannot be passed directly to the kernel and must be copied to the GPU's DRAM. Thus, buffers for light position and transformation matrices are allocated. Finally, the mesh data is copied to two constant buffers.

The primitive assembly initialization allocates a constant index buffer and copies the corresponding data. A buffer for triangles is allocated to be filled on each draw call.

The raster operation stage obtains a pointer to the actual window frame buffer the window system initialization created. The raster operation initialization then allocates a frame buffer on the GPU of the same size as the window frame buffer.

The pixel shader initialization is called with pointers to the frame buffer and various light-related data. A fragment buffer is allocated with the exact resolution as the frame buffer. Since the data of the light used in the pixel shader does not change throughout the execution of the application, corresponding buffers are allocated, and the data is copied right away.

The rasterization is initialized with pointers to the triangle buffer and a fragment buffer of the pixel shader. Additionally, two queue buffers are initialized for triangle binning and tiling.

Finally, the render thread enters the render loop, periodically checking if the window is about to close and performing the draw call otherwise. The execution of the pipeline is timed, and if it takes less than approximately 16 and a half milliseconds, the render thread sleeps for the remainder of the time.

```
/* render_procedure */

while (window_get_status() == WINDOW_ACTIVE) {
    timer_update(&timer);

    // Draw call
    ...

    // Sleep
    timer_update(&timer);
    thread_sleep(target - timer.elapsed);
}
```

5.2.2 Cleanup

While the main thread processes window events and updates the scene accordingly, it will eventually register a request to terminate the application. When this happens, the window is marked with a flag that it is about to close. The main thread leaves the event processing loop and waits for the render thread to terminate.

The render thread finishes the current draw call, renders the final frame buffer to the screen, and then leaves the render loop. All the pipeline stages are cleaned up in the opposite order they were initialized, freeing the buffers they allocated on the GPU. Lastly, the mutex guarding the scene data access is destroyed, and the mesh is unloaded from the host memory.

After the render thread finishes its execution, the main thread frees the window, timer, and print systems, finally terminating the application. All the initialization, memory management, and cleanup are checked for errors. Additionally, each pipeline stage and each kernel launch are checked for errors.

5.3 Scene Fetch

The primary purpose of the scene fetch stage is to construct the model, view, and perspective transformation matrices and combine them into a single one used in the vertex shader to transform vertices.

5.3.1 Model Matrix

The object in the scene is configured with a position, rotation, and scale. The purpose of the model matrix is to combine three linear transformation matrices of translation, rotation, and scale into a single one, which is then applied to each vertex through matrix multiplication. While the mesh can be rotated and scaled with a linear transformation by multiplying the corresponding 3×3 matrix with the three-dimensional mesh vertex vector, the movement of vertices cannot be represented in such a way.

Computer graphics applications extend the three-dimensional cartesian coordinates of vectors to four-dimensional homogenous coordinates by appending a value called weight. When this value equals 0, the four-dimensional vector of homogenous coordinates represents a direction in the three-dimensional space, while the value of 1 is used for positions. The extension to homogenous coordinates allows representing the translation transformation as a linear transformation with a 4×4 matrix. All three translation, rotation, and scale matrices can then be combined into one. Therefore, all the transformations of the object vertices use the homogenous coordinate system [9]. The model matrix has the following format:

$$M_T * M_R * M_S$$

M_S is the scale matrix:

$$\begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

M_R is the rotation matrix, first applying the rotation around the x-axis, then y-axis, and finally the z-axis:

$$R_Z * R_Y * R_X$$

M_T is the translation matrix, translating by t_x, t_y, and t_z:

$$\begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

```
/* scene_fetch */

// Construct model matrix
mat3_t mat = { 0 };
mat3_identity(mat);
mat3_scale_xyz(scene.model_scale, mat);
mat3_rotate_xyz(scene.model_rotation, mat);
mat3_to_mat4(mat, model); // Pointer to the matrix in vertex shader
mat4_translate_xyz(scene.model_position, model);
```

5.3.2 View Matrix

The vital task of vertex processing is to convert the world screen coordinate system to the viewing coordinate system with a camera at its origin. The camera's viewing direction vector is obtained by subtracting the camera position from the position the camera is looking at. This vector must be projected to $[0, 0, -1, 0]$. The camera's orientation and, thus, its rotation around the view vector is specified by an up vector, which always points up in the world coordinate system of the thesis implementation. The last vector pointing to the right is obtained by multiplying the viewing and up vectors. All three vectors are then normalized and serve as a basis of the viewing coordinate system. Finally, the change of coordinate system base has to be followed by a translation so that the camera lies at the origin [9]. The matrix has the following format:

$$\begin{pmatrix} p_x & p_y & p_z & -p \cdot eye \\ h_x & h_y & h_z & -h \cdot eye \\ -l_x & -l_y & -l_z & -l \cdot eye \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

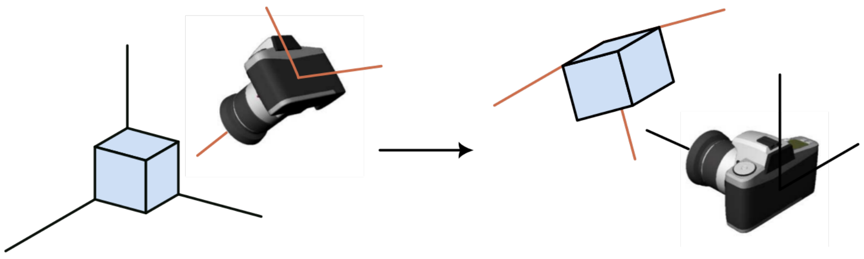
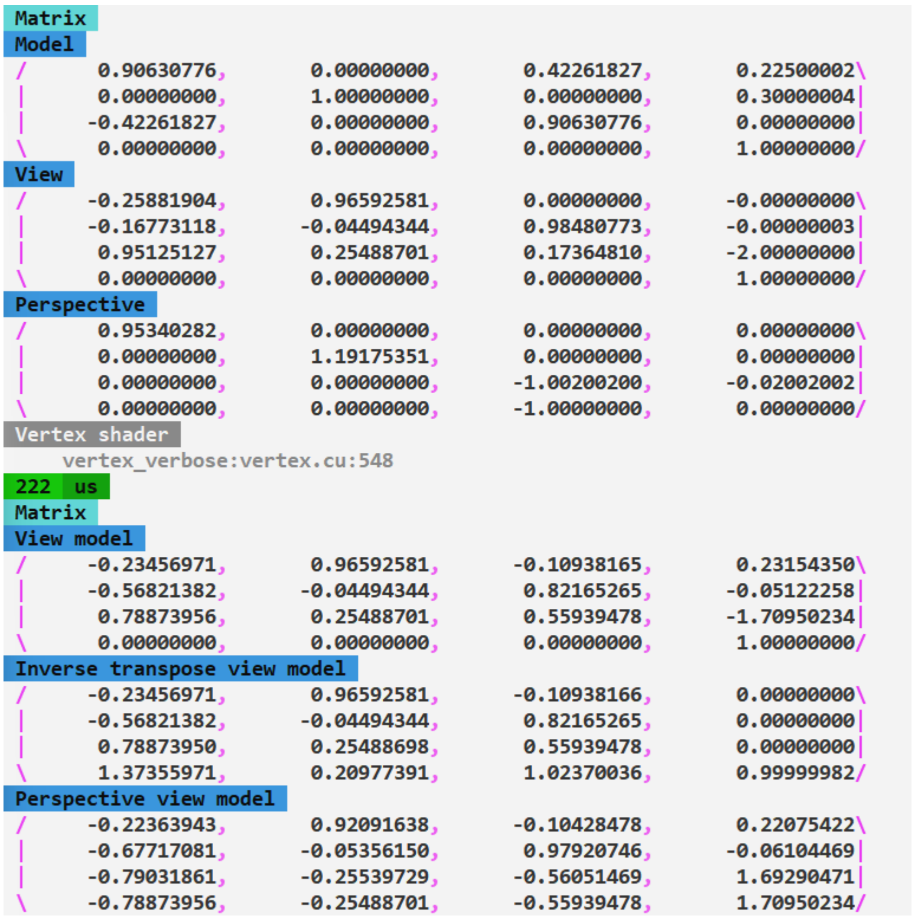


Figure 5.2. The conversion from the world coordinates to viewing coordinates. The camera becomes a new world origin, with its vectors becoming the new basis. The camera image is taken from [9].

```
/* scene_fetch */

// Construct view matrix
vec3_t eye, center, up;
mat4_t mat;

vec3_t l = { 0 }; // The camera viewing direction vector
vec3_subtract(center, eye, l); // l = center - eye
vec3_normalize(l);

vec3_t p = { 0 }; // The right vector
vec3_cross(l, up, p); // The up vector aims up in the world space
vec3_normalize(p);

vec3_t h = { 0 }; // The actual camera space up vector
vec3_cross(p, l, h);

// Fill the matrix with values
...
```

5.3.3 Perspective Matrix

The perspective projection models the proportional shrinking of the object when the distance from the camera increases and provides a spatial perception of the 2D screen. The purpose of the perspective matrix is to convert the vertex view coordinates to clip coordinates by transforming vertices inside a view frustum into a unit cube. Such coordinates will be used to perform view frustum clipping and calculate normalized device coordinates. The view frustum is an area before the camera bounding the vertices that should be projected on the screen. Two elemental planes of the view frustum are near and far clipping planes discarding vertices closer or farther from the camera than



```
Matrix
Model
/      0.90630776,       0.00000000,       0.42261827,       0.22500002\
|      0.00000000,       1.00000000,       0.00000000,       0.30000004|
|     -0.42261827,       0.00000000,       0.90630776,       0.00000000|
\      0.00000000,       0.00000000,       0.00000000,       1.00000000/
View
/     -0.25881904,       0.96592581,       0.00000000,      -0.00000000\
|     -0.16773118,      -0.04494344,       0.98480773,      -0.00000003|
|      0.95125127,       0.25488701,       0.17364810,      -2.00000000|
\      0.00000000,       0.00000000,       0.00000000,       1.00000000/
Perspective
/      0.95340282,       0.00000000,       0.00000000,       0.00000000\
|      0.00000000,       1.19175351,       0.00000000,       0.00000000|
|      0.00000000,       0.00000000,      -1.00200200,      -0.02002002|
\      0.00000000,       0.00000000,      -1.00000000,       0.00000000/
Vertex shader
     vertex_verbose:vertex.cu:548
222  us
Matrix
View model
/     -0.23456971,       0.96592581,      -0.10938165,       0.23154350\
|     -0.56821382,      -0.04494344,       0.82165265,      -0.05122258|
|      0.78873956,       0.25488701,       0.55939478,      -1.70950234|
\      0.00000000,       0.00000000,       0.00000000,       1.00000000/
Inverse transpose view model
/     -0.23456971,       0.96592581,      -0.10938166,       0.00000000\
|     -0.56821382,      -0.04494344,       0.82165265,       0.00000000|
|      0.78873950,       0.25488698,       0.55939478,       0.00000000|
\      1.37355971,       0.20977391,       1.02370036,       0.99999982/
Perspective view model
/     -0.22363943,       0.92091638,      -0.10428478,       0.22075422\
|     -0.67717081,      -0.05356150,       0.97920746,      -0.06104469|
|     -0.79031861,      -0.25539729,      -0.56051469,       1.69290471|
\     -0.78873956,      -0.25488701,      -0.55939478,       1.70950234/
```

Figure 5.3. Verbose output of all the transformation matrices.

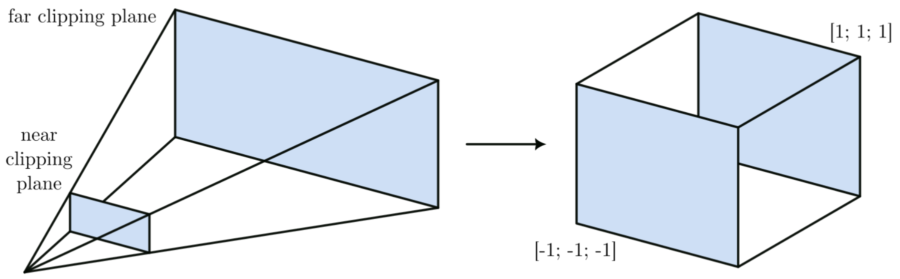


Figure 5.4. The conversion of the vertex view coordinates inside the view frustum to clip coordinates.

the near or far clipping plane, respectively [9]. The implementation of the perspective matrix is taken from the cglm library [39]. Finally, all the matrices are multiplied to obtain the view model and perspective view model. Inverse transpose view model is needed to correctly transform normals.

5.4 Vertex Shader

Two kernels are executed. The vertex kernel uses view, view model, and perspective view model matrices together with the light position. The normal vector kernel uses inverse transpose of the view model matrix.

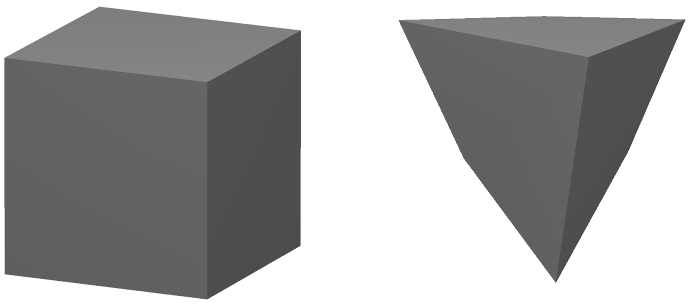


Figure 5.5. The difference of the perspective when changing the vertical field of view. The left view has a FOV set to 1 degree, while the right is set to 110 degrees.

The kernel that processes vertices and generates vectors for the Blinn-Phong illumination model used in the later pipeline stage is launched with several blocks, each with a single warp of 32 threads, with each thread processing one input vertex. First, the light and vertex position in viewing coordinates is calculated by multiplying them with the view and view model matrices. The light direction vector is obtained by subtracting the vertex position from the light position, and the half vector is obtained by the exact subtraction repeated. Vectors are normalized and saved to the output vertex structure inside a pre-allocated buffer. The position of each vertex is obtained by multiplying the input vertex position with the perspective view model transformation matrix. Finally, the perspective division is performed by dividing the x, y, and z components with the weight value, which will generally not be the original 1, transforming the four-dimensional clip space to three-dimensional normalized device coordinates.

The kernel that processes the vertex normals is launched with the same configuration of 32 threads per block and enough blocks to process all the data. The number of normals may differ from the number of vertices, thus the separate kernel. Since the scale matrix component of the model matrix is not guaranteed to be uniform across all the axis, normals are transformed with an inverse transpose of the view model matrix to work correctly. As with the light and half vectors, the normal vector is also not transformed with the perspective matrix. A synchronization barrier is used at the end of the vertex shader.

```
/* vertex_shader */

unsigned int warps = size_vertices / 32;
vertices_kernel <<< warps, 32 >>> (
    view,
    view_model,
    perspective_view_model,
    size_vertices,
    const_vertices, // Buffer of reference mesh vertices
    vertices,       // Buffer to save transformed vertices
    light_position);

unsigned int warps = size_normals / 32;
normals_kernel <<< warps, 32 >>> (
    view_model_inverse_transpose,
    size_normals,
      const_normals, // Buffer of reference mesh normals
      normals);      // Buffer to save transformed normals
```

```
/* vertices_kernel */

vec4_multiply_mat(view, light_vector, light_vector);
vec4_multiply_mat(view_model, const_pos, transformed_pos);
vec4_subtract(light_vector, transformed_pos, light_vector);
vec4_subtract(light_Vector, transformed_pos, half_vector);

// vec4 to vec3

vec3_normalize(light_vector);
vec3_normalize(half_vector);

vec4_multiply(perspective_view_model, const_pos, transformed_pos);

// transformed_pos.xyz / transformed_pos.w
```

```
/* normals_kernel */

vec4_multiply_mat(inverse_transpose, const_norm, transformed_norm);

// vec4 to vec3

vec3_normalize(transformed_norm);
```

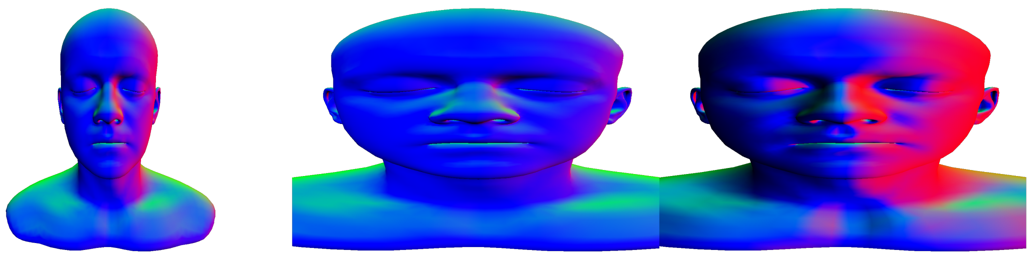
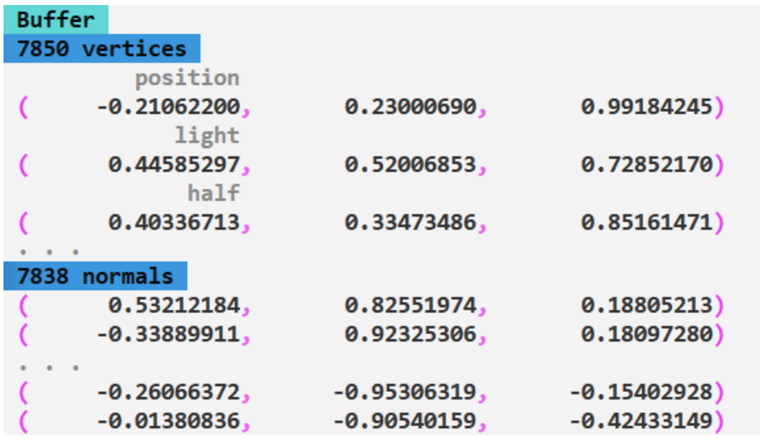


Figure 5.6. Head object with normal vectors color-coded to the rendered pixels illustrates the incorrect multiplication of the normal vector by the view model matrix visible on the right head. The middle head multiplies the normal vectors with the correct inverse transpose view model matrix, with the vectors keeping the correct headings when compared to the original unscaled mesh on the left. The head was scaled by a factor of 3 on a single axis.

5.5 Primitive Assembly

A kernel is launched with several blocks, each with a single warp of 32 threads and each thread processing a single input triangle. First, the kernel copies and groups the data from the input vertices into a buffer of triangle structures. The vertex index is taken



```
Buffer
7850 vertices
            position
(      -0.21062200,        0.23000690,        0.99184245)
               light
(       0.44585297,        0.52006853,        0.72852170)
                half
(       0.40336713,        0.33473486,        0.85161471)
. . .
7838 normals
(       0.53212184,        0.82551974,        0.18805213)
(      -0.33889911,        0.92325306,        0.18097280)
. . .
(      -0.26066372,       -0.95306319,       -0.15402928)
(      -0.01380836,       -0.90540159,       -0.42433149)
```

Figure 5.7. Verbose output of the vertex shader.

from an index buffer loaded from the .obj file. After the triangles are formed, they undergo a series of culling steps.

```
/* primitive_assembly */

unsigned int warps = size_triangles / 32;
kernel <<< warps, 32 >> (
    const_vertices, // Transformed vertices
    const_normals,  // Transformed normals
    const_indices,  // Triangle indices
    size_triangles,
    triangles);     // Buffer to save assembled triangles
```

```
/* kernel */

assemble_triangle(vertices, normals, indices); // Only copies data
backface_culling(triangle);
view_frustum_clipping(triangle);
clipspace_to_screenspace(triangle);
degenerate_culling(triangle);
```

```
/* clipspace_to_screenspace */

// For all three triangle vertices
pos.x = (pos.x + 1) * 0.5 * window_width;
pos.y = (pos.y + 1) * 0.5 * window_height;
pos.z = (pos.z + 1) * 0.5;
```

5.5.1 Back-Face Culling

Each triangle has a facing orientation based on the order of vertices, which can be clockwise or counterclockwise. The rendering pipeline chooses one of the orientations as the front-facing one, discarding the triangles with the other. For meshes with a volume and no holes, back-facing triangles are guaranteed to be occluded by the front-facing triangles closer to the camera. The triangle's orientation can be determined by taking two vectors coming out from one of the vertices, each representing one edge

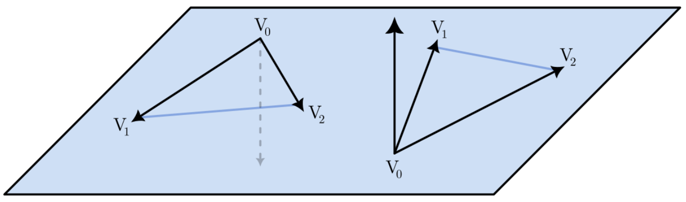


Figure 5.8. The change of the orientation of triangle vertices results in the change of orientation of the resulting vector obtained from the cross product.

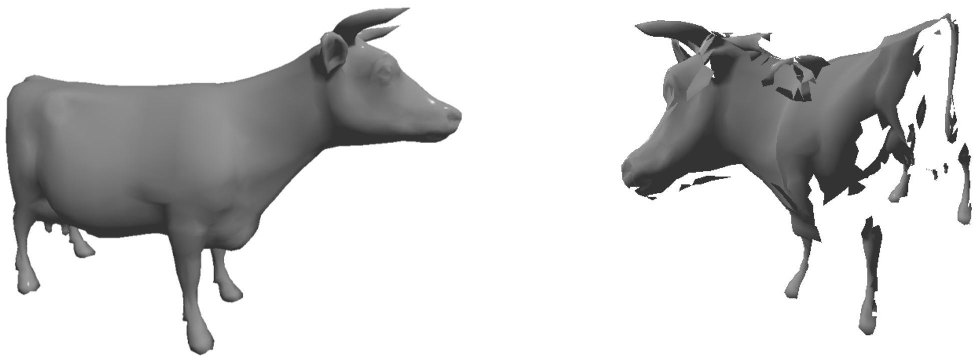


Figure 5.9. When disabling back-face culling with a shortcut, keeping the culled triangles culled until enabled again, the rotation of the camera on the right reveals which part of the cow object was back-face culled when viewing from the angle on the left.

of the triangle, setting their third coordinate to 0, and then computing a cross product. The sign of the third component of the resulting vector determines the orientation.

```
/* backface_culling */

// Edge vectors
edge1.x = v2.x - v1.x;
edge1.y = v2.y - v1.y;
edge2.x = v3.x - v1.x;
edge2.y = v3.y - v1.y;

if (edge1.x * edge2.y - edge1.y * edge2.x < 0) {
    // Cull
}
```

5.5.2 View-Frustum Clipping

Although the function is called a clipping in the implementation, no actual clipping happens. A three-dimensional bounding box is computed, defined by two points with minimum and maximum x, y, and z components of the processed triangle. It is then checked whether the bounding box intersects the volume of normalized device coordinates. If the triangle lies entirely off the volume, it is discarded. However, if at least one of the vertices resides inside the volume, the triangle is passed to the rasterization.

Additionally, if a triangle has at least one vertex closer to the camera than the near clipping plane or farther from the camera than the far clipping plane, it is culled. A visually more pleasing solution would be a proper clipping, where the triangle is split at the point of the intersection with one of the clipping planes. However, such a technique may generate more triangles than were inputted, needing to dynamically reallocate the triangle buffer or leave space for such triangles on the buffer.

```
/* view_frustum_clipping */

// Get 3D bounding box
triangle_aabb_3d(triangle, min, max);

if (min.x > 1 || min.y > 1 || max.x < -1 || max.y < -1 ||
    min.z < 0 || min.z > 1 || max.z < 0 || max.z > 1) {
    // Cull
}
```

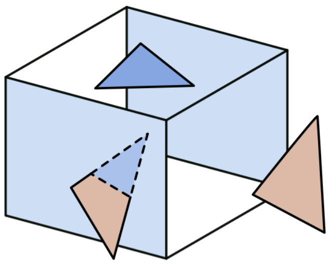


Figure 5.10. Triangles completely outside the view frustum are culled, and the ones intersecting a wall of the unit cube are clipped.

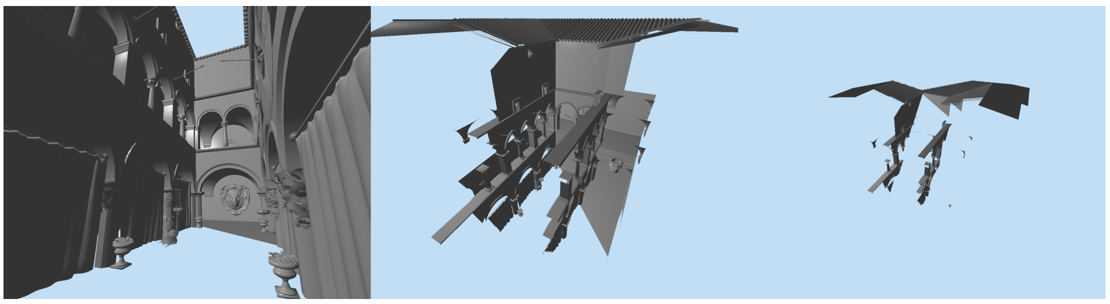


Figure 5.11. Sponza scene is a perfect showcase of the view-frustum clipping. When disabling the clipping from the angle on the left and zooming out, the middle view shows the portion of the mesh that was culled by the near clipping plane. Zooming even farther causes the rest of the mesh to be clipped by the far clipping plane.

5.5.3 Degenerate Culling

Degenerate culling happens after transforming the clip space to the screen space, converting the unit range scale of the x and y components to the screen position and the z component to a depth value. An area of the triangle is calculated, and if it is too small, the triangle is culled. Such a technique helps discard triangles smaller than a pixel or rotated so that their area is too small, which would not otherwise contribute to the rendered screen, wasting the queue buffer space in the rasterization stage.

```
/* degenerate_culling */

float area = (v1.x * v2.y + v2.x * v3.y + v1.y * v3.x -
              v3.x * v2.y - v1.y * v2.x - v1.x * v3.y) * 0.5;
if (-degenerate_factor < area && area < degenerate_factor) {
    // Cull
}
```

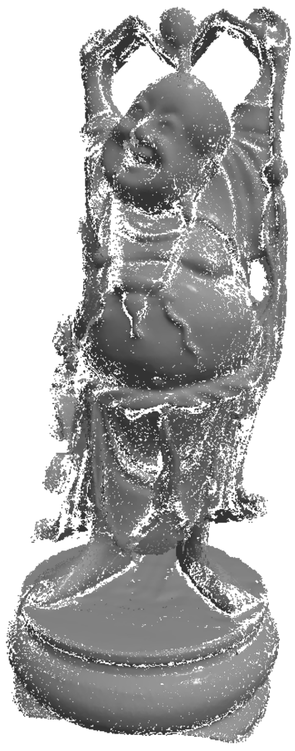
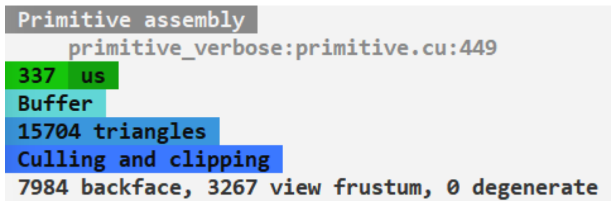


Figure 5.12. The degenerate factor of 0.5 is clearly too large for the buddha model.



```
Primitive assembly
    primitive_verbose:primitive.cu:449
337  us
Buffer
15704 triangles
Culling and clipping
7984 backface, 3267 view frustum, 0 degenerate
```

Figure 5.13. Verbose output of the primitive assembly.

5.6 Rasterization

The rasterization stage has two GPU buffers that must be zeroed out on each pipeline render pass. They contain triangle queues and counters. The stage works by sorting the input triangles into bins, then sorting the triangles from the bins into tiles, and then rasterizing the triangles from the tiles. Each tile covers a 16×16 pixel area on the screen, with each bin consisting of 8×8 tiles. The default screen resolution is set to 1,280×1,024 pixels. Such configuration ensures everything is a multiple of 32, enabling the GPU to run most efficiently with the warps execution model. The rasterization stage is the bottleneck, which was expected before the implementation. There are multiple barrier synchronizations.

```
/* rasterization */

dim3 bin_blocks(bin_resolution.x, bin_resolution.y);
bin_kernel <<< bin_blocks, 1024 >>> (
    size_triangles,
    triangles,
    bins); // Bin queues

dim3 tile_blocks(tile_resolution.x, tile_resolution.y);
bin_kernel <<< tile_blocks, 1024 >>> (
    triangles,
    bins,
    tiles); // Tile queues

dim3 tile_blocks(tile_resolution.x, tile_resolution.y);
rasterize_kernel <<< tile_blocks, tile_size * tile_size >>> (
    triangles,
    tiles,
    fragments); // Fragment buffer
```

5.6.1 Binning and Tiling

The bin kernel is launched with block configuration corresponding to the number of bins splitting the screen, with each block running the maximum permitted number of 1,024 threads [38]. The kernel sets up a variable in a shared memory that threads can only access within a block. This variable synchronizes the actual number of triangles in the queue buffer between threads and provides an exclusive index to the buffer. Incrementing of the counter variable is atomic, and no data races happen between threads.

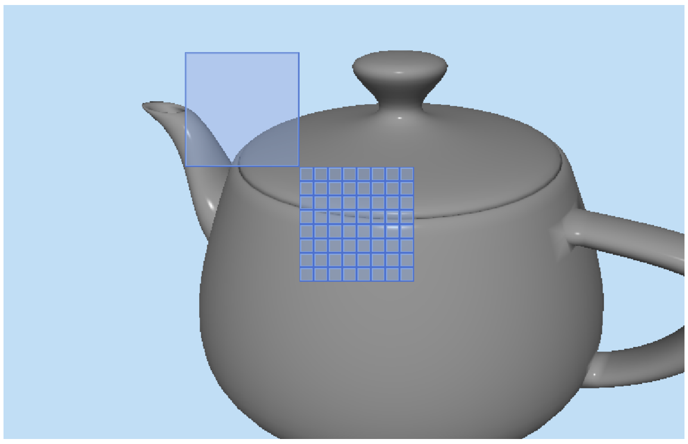


Figure 5.14. By default, the screen is split into 10×8 bins, each split into 8×8 tiles, each occupying an area of 16×16 pixels.

The thread evaluates how many triangles it should process and then loops through them, calculating their 2D bounding box. If the triangle overlaps the bin at hand and is not culled, the bin counter variable is incremented, with the atomic operation returning the variable's value. This value is used to index into the bin buffer. The total count of triangles in the current bin queue is saved as the first element of the queue, followed by triangle indexes in the triangle buffer. The count may increment above the maximum capacity of the queue, and the user is notified that a bin or a tile overflow happened. Both queues have fixed sizes and do not adapt dynamically to the rendered scene.

```
/* bin_kernel */

__shared__ unsigned int count;
__syncthreads();

// Compute x_min, y_min, x_max, y_max of the bin

unsigned int batch = (size_triangles / 1024) + 1;
unsigned int current = 0;
while (current < batch) {
    if (triangle not culled) {
        // Get 2D bounding box
        triangle_aabb_2d(triangle, min, max);
        if (min.x > x_max || min.y > y_max ||
            max.x < x_min || max.y < y_min) {
        } else {
            int i = atomicAdd(&count, 1);
            if (i < buffer limit) {
                bin[corresponding index] = triangle_id;
            }
        }
    }
}

__syncthreads();
bin[first element of the queue] = count;
```

The tile kernel is launched with the same configuration of the GPU's maximum of 1,024 threads per block, with the number of blocks corresponding to the number of tiles on the screen. First, the index of the bin buffer the tile should process must be obtained since the kernel is launched with a grid of blocks, indexing the tiles by rows and columns, which does not correspond to the actual conceptual screen splitting. When the index is evaluated, the thread calculates the number of triangles it should process and then iterates through the queue of triangles, splitting them further into tile queues. Again, the first element of each tile buffer contains the total triangle count assigned to that tile.

```
/* tile_kernel */

// Equivalent to bin_kernel
```

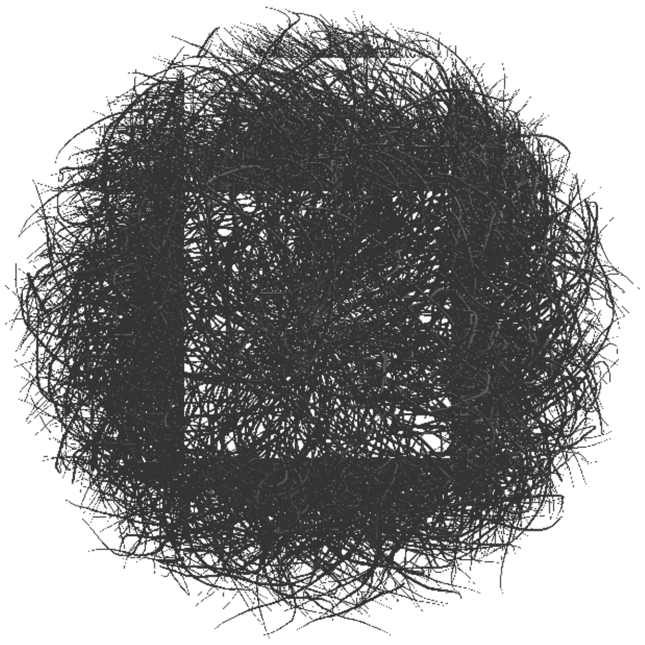
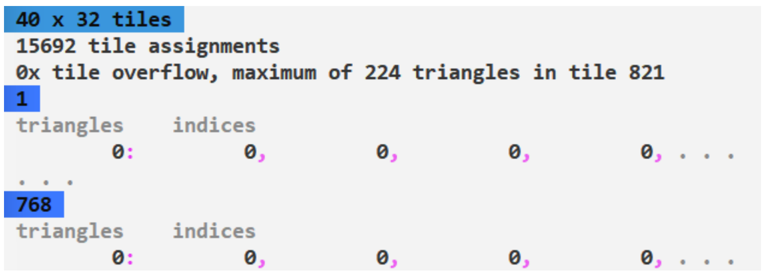


Figure 5.15. The binning rasterization kernel with a queue buffer size of 30,000 does not fit the overlapping subset of 2,880,000 triangles of the hairball mesh. The user is notified of the bin and tile overflow when printing the verbose output to the terminal, since the overflow happening is not always evident. In this case, however, the overflow is clearly visible, revealing the borders of some of the screen bins. When the queues overflow, the corresponding part of the screen flickers on each draw call due to all the threads battling for the limited space of the queue.



```
40 x 32 tiles
15692 tile assignments
0x tile overflow, maximum of 224 triangles in tile 821
1
triangles    indices
        0:          0,         0,         0,         0, . . .
. . .
768
triangles    indices
        0:          0,         0,         0,         0, . . .
```

Figure 5.16. Verbose output of the rasterization.

5.6.2 Fragments

When the triangles are sorted, the rasterization stage enters its last kernel, which rasterizes them into fragments. The kernel is launched with the exact number of blocks as the tile kernel, with the number of threads in each block corresponding to the number of pixels in each tile. The x and y coordinates of the middle of the fragment are evaluated and used to compute the barycentric coordinates inside the triangle. The algorithm works by constructing three vectors with an origin at one of the vertices, two representing the triangle edges, with the third being the vector from the vertex to the reference point. Dot products between vectors are calculated and used to obtain the intersection point. The condition guards against computing barycentric coordinates for triangles almost parallel with the ray casted from the middle of the fragment, which produces visual artifacts if not handled. The algorithm is inspired by Möller–Trumbore intersection algorithm [43].

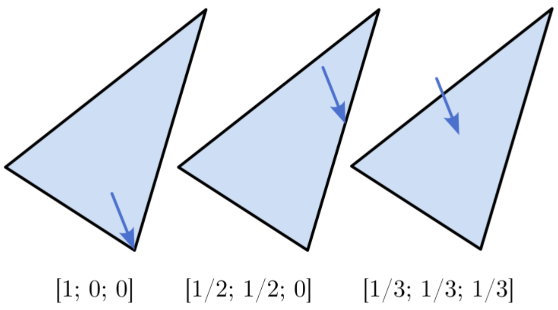


Figure 5.17. Barycentric coordinates.

When the barycentric coordinates are obtained, the thread checks if the point lies inside the triangle. If the test passes, a depth test is performed, comparing the depth of the intersecting point on the triangle with the minimum depth registered for the current fragment. The barycentric coordinates are then used as weights to interpolate the data of the three vertices for the resulting fragment. Normal, light, and half vectors are interpolated, normalized, and saved to the fragment buffer.

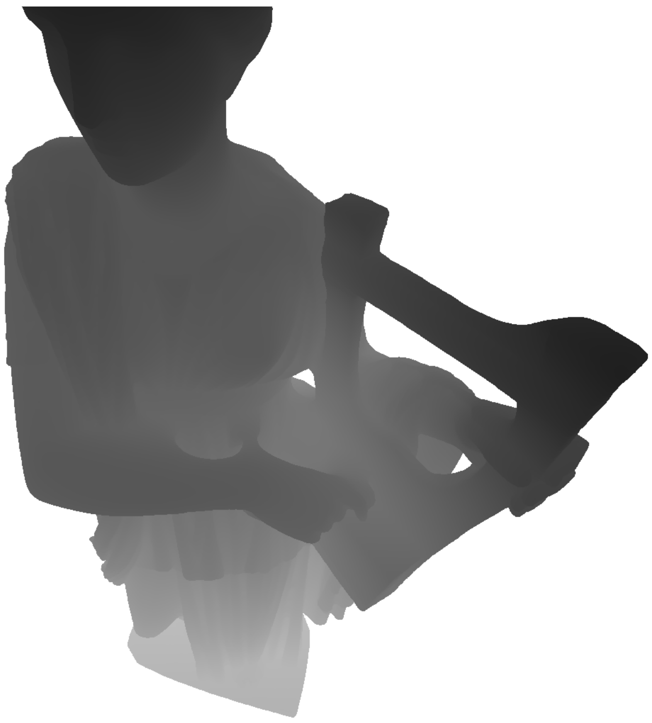


Figure 5.18. Each pixel of erato mesh is assigned a grayscale color based on the distance from the camera, with darker pixels being closer.

```
/* rasterize_kernel */
```

```
// Compute x and y of the fragment

float min_depth = 1.0;

if (cartesian_to_barycentric(triangle, point, barycentric) {
    if (barycentric.x >= 0 && barycentric.x <= 1 &&
        barycentric.y >= 0 && barycentric.y <= 1 &&
        barycentric.z >= 0 && barycentric.z <= 1 &&
        depth < min_depth) {
        // Inside triangle and closer to the camera
        min_depth = depth;

        normal = normal1 * barycentric.x + normal2 * barycentric.y ...
        light = ...
        half = ...

        vec3_normalize(normal);
        vec3_normalize(light);
        vec3_normalize(half);

        // Assign normal, light, and half vectors to the fragment
    }
}
```

5.7 Pixel Shader

The single kernel of the pixel shader is launched with 16×16 threads in each block and the corresponding number of blocks to process the entire fragment buffer. The purpose of this stage is to compute the Blinn-Phong illumination model for each fragment and save the corresponding pixel to the device frame buffer. The kernel computes ambient, diffuse, and specular components of the illumination model [9] and their contribution to the resulting pixel.

```
/* pixel_shader */

dim3 blocks(window_width / 16, window_height / 16);
dim3 threads(16, 16);
kernel <<< blocks, threads >>> (
    fragments,
    framebuffer,
    light_constant, // Weights to multiply light components
    light_ambient,
    light_diffuse,
    light_specular,
    light_shininess);
```

The kernel begins its operation by checking whether a normal vector is assigned to the fragment. If not, no geometry overlaps the fragment, and the pixel is not shaded. Since the ambient light is visible the same way from any view direction, the kernel only needs to copy the ambient color and multiply it with the corresponding component weight. The diffuse component is multiplied by the weight and dot product of the normal and light vectors, which is clamped not to be negative. The specular component is multiplied by the weight and dot product of the half and light vectors clamped the same. The resulting color, a sum of these three light components, is saved to the corresponding pixel in the device frame buffer.

```
/* kernel */

if (normal != 0) {
    // Compute dot products
    float normal_light, normal_half;
    normal_light = vec3_dot(normal, light);
    normal_light = maximum(normal_light, 0);
    normal_half = vec3_dot(normal, half);
    normal_half = maximum(normal_half, 0);

    // Set color
    color += ambient * constant[0];
    color += diffuse * constant[1] * normal_light;
    color += specular * constant[2] * pow(normal_half, shininess);
}
```

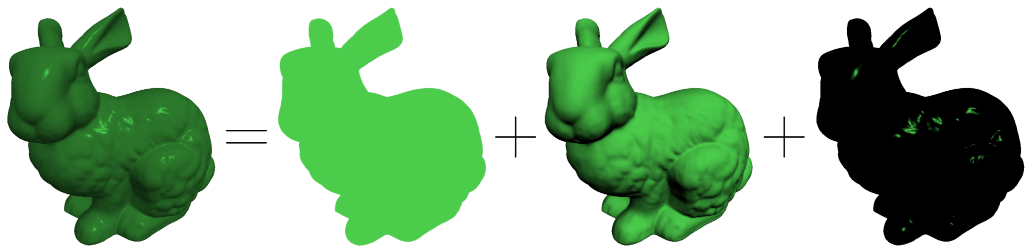


Figure 5.19. Sum of the weighted light components produces the lighting effect. The default weight values are 0.2, 0.4, 0.4 for ambient, diffuse, and specular, respectively.

5.8 Raster Operation

The final pipeline stage does not need to launch any kernel. It copies the device frame buffer back to the host, setting the bitmap's pixels spanning the window frame. The stage then invalidates the screen, raising a redraw event picked up by the main thread event processing loop. The draw call is finished. The render thread sleeps for the remainder of the frame time and then performs the next render pass, repeating the computations in all the pipeline stages.

```
/* raster_operation */

InvalidateRect(window.handle);
```

```
/* window_draw_callback */

BitBlt(bitmap, framebuffer);
```

Chapter 6 Evaluation

The thesis supports two build configurations. The debug build is used to evaluate the visual correctness of the implementation. It provides verbose debug output and different rendering modes, while the release build is used to evaluate the actual performance of the pipeline. Optimizations used are described in the source code appendix. The implementation provides three benchmarks the user can run to test the performance of the pipeline on his system. The results are saved to a text file, and a Python script is provided to plot the data. The source code includes all the measured data presented in this chapter. The pipeline was evaluated on objects frequently used by researchers, which are freely available to the user to download [36]. Only some smaller objects are included in the source.

6.1 Target Platform

The performance of the pipeline is tested on two different systems. The performant high-end system runs with a GeForce RTX 3070 graphics card with a GA104 chip based on NVIDIA's Ampere microarchitecture. First released in the second half of 2020, the GPU with 17,400 million transistors is manufactured using an 8-nanometer production process on a die of 392 square millimeters. It features 5,888 programmable CUDA cores grouped in 48 SMs, 184 texture mapping hardware-accelerated units (TMUs), 96 ROPs, 184 tensor cores, and 46 ray tracing cores. Each SM has a level-1 cache of 128 kilobytes and can access a level-2 cache of 4,096 kilobytes. The maximum thermal design power (TDP) is 220 watts. RTX 3070 has 8 gigabytes of DRAM and a 256-bit wide memory lane. The theoretical floating point performance is 20,310 GigaFLOPS. The GPU has CUDA compute capability of 8.6 [22].

On the opposite spectrum of graphical performance, the second test system runs with a GeForce MX250 mobile graphics card based on Pascal architecture GP108 chip. The GPU released at the beginning of 2019 is manufactured with a 14-nanometer technology on a die of 74 square millimeters with 1,800 million transistors. It features 384 CUDA cores in 3 streaming multiprocessors, 24 TMUs, and 16 ROPs. Each SM has 48 kilobytes of level-1 cache and access to 512 kilobytes of level-2 cache. The 2 gigabytes of DRAM are connected to the GPU through a 64-bit memory bus. With a TDP of 25 watts, the theoretical floating point performance of the MX250 is 1,215 GigaFLOPS. Theoretically, the pipeline should perform 20 times better on the RTX 3070 GPU. MX250 supports CUDA CC of 6.1 [23].

A benchmark on yet another graphics card, the NVIDIA GeForce GTX 1080, was evaluated to compare the performance of the thesis pipeline to the cuRE [3] implementation. The research article includes a table with the average draw call time tested on the GTX 1080 GPU.

6.2 Correctness

The pipeline implementation rarely experiences issues with pixels not being part of the mesh being shaded. This behavior is probably caused by dividing with a small enough number in the computation of the barycentric coordinates. The value is already checked to not divide by zero or a tiny number, but more is needed. Further experimentation is required.

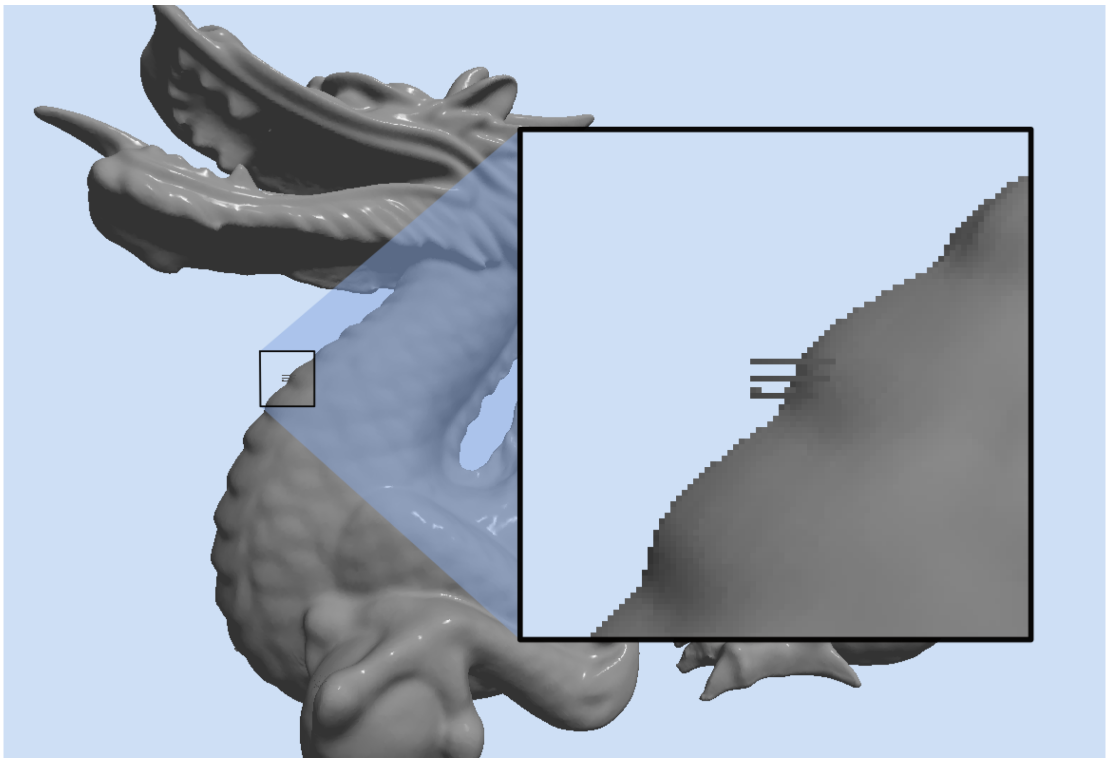


Figure 6.1. Small artifact when rendering dragon object.

6.3 Benchmarks

Three benchmarks are implemented in the code, with each of the six stages timed. The idle test records the execution of 300 idle frames. The results of this benchmark are not presented in the text but can be found in the source. This simple benchmark was used throughout the development to measure the fluctuations in the execution of the graphics pipeline between frames. The second benchmark evaluates the performance of a 360 degrees camera rotation around the z-axis across 720 frames. The third benchmark measures the execution of camera zoom, starting with the camera close to the origin and most of the mesh being culled by the near clipping plane and progressively zooming out until the mesh occupies approximately a single bin.

The benchmarks were evaluated on six increasingly more complex scenes. Objects that could be converted to the suitable application-supported .obj format were selected from the archive [36], namely the cube, white oak, bunny, erato, and hairball. The source code additionally includes the cow model.

Lastly, several 360 degrees camera rotation benchmarks were conducted with varying resolutions of bins, tiles, and the window itself. The measured data presents a varying performance of the rasterization stage. All the results are discussed, and the pipeline bottlenecks are identified.

Object	Vertices	Triangles
Cube	8	12
Cow	2,903	5,804
White Oak	38,199	36,760
Bunny	72,027	144,046
Erato	1,237,495	412,498
Hairball	1,470,000	2,880,000

Table 6.1. Mesh data statistics of benchmarked objects.

6.3.1 Camera Rotation

The performance of the pipeline is expected to be consistent throughout the evaluation of this benchmark. The object stays in the middle of the window, and the variation in bin and tile triangle occupation is minimal. The sizes of the bin and tile queues are hardcoded in the release build, with $2^{19} = 524{,}288$ capacity of the bin queue and $2^{17} = 131{,}072$ tile capacity. Although such an enormous size of buffers is wasting space for the small models, it is required for the hairball model to be displayed correctly without bins and tiles overflowing. Each tile occupies 16×16 pixels, and each bin constitutes 8×8 tiles. The screen resolution is 1,280×1,024 pixels, 10×8 bins, and 80×64 tiles.

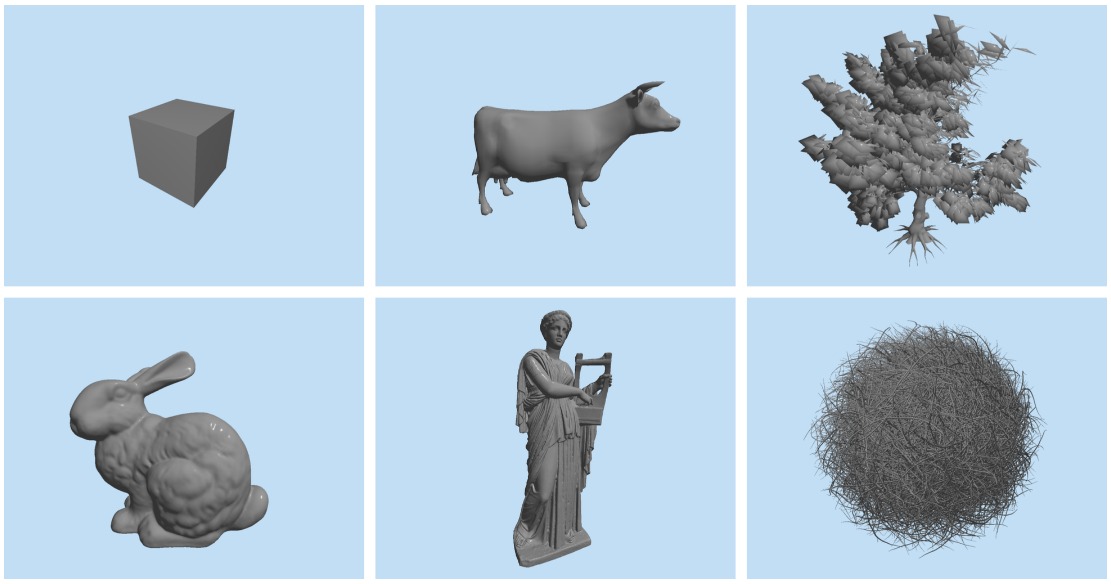


Figure 6.2. Initial scene configurations for 360 degrees camera rotation benchmark.

As expected, the rasterization stage is the main bottleneck of the pipeline, dominating the execution time. The time needed to compute the other stages is negligible. The graph of the erato 360 degrees camera rotation benchmark clearly illustrates this. The average execution time of the pipeline is 14.9 milliseconds, just keeping within the target of 60 frames per second, with an average of 13 milliseconds spent in the rasterization stage. Other stages can be distinguished by using a logarithmic scale on the y-axis of the graph and are color-coded base on their function. The red-colored vertex shader and primitive assembly stages perform geometry processing, while the green-colored pixel shader and raster operation stages perform handling of the rasterized fragments and pixels. A dotted line was used to plot the data of the scene fetch and raster operation stages that do not launch any kernel and thus do not run on the GPU.

The scene fetch is executed the fastest out of all pipeline stages, with an average of 1 microsecond. The following stage in the pipeline, the vertex shader, executes in 0.3 milliseconds on average. The GPU computes the primitive assembly stage almost for a whole millisecond, possibly due to memory accesses into multiple buffers in the DRAM and copying the data when forming triangles. The rasterization stage is followed by the pixel shader stage with 0.2 milliseconds execution time on average. Finally, the raster operation stage finishes the pipeline with 0.4 milliseconds, with its execution time consistent across all the workloads and benchmarks.

RTX 3070	Cube	Cow	White Oak
Average (μs)	8,259	8,110	9,119
Std (μs)	301	152	225
Relative (%)	3.65	1.88	2.47
	Bunny	**Erato**	**Hairball**
Average (μs)	9,726	14,875	135,657
Std (μs)	276	780	2,927
Relative (%)	2.84	5.25	2.16

Table 6.2. Measurements of total pipeline execution times of 360 degrees camera rotation benchmark on RTX 3070 GPU. Averages and standard deviations are in microseconds, and the relative percentage is obtained by dividing the standard deviation by the average.

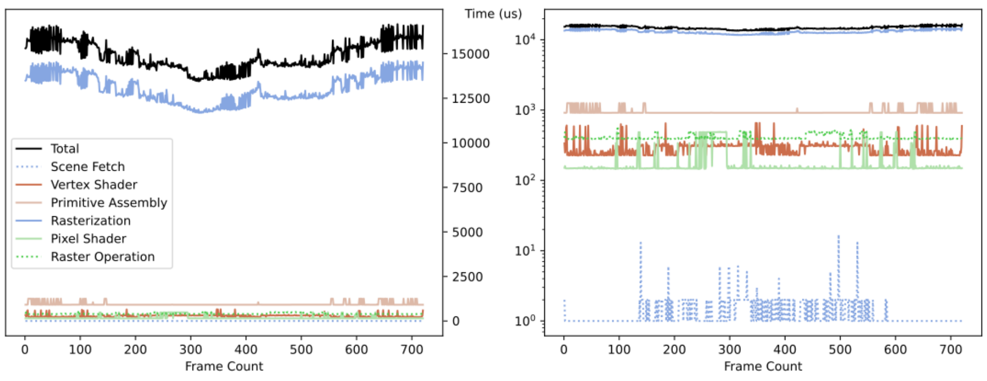


Figure 6.3. Erato 360 degrees camera rotation benchmark results on the RTX 3070 GPU. Logarithmic scale for the y-axis is used for the plot on the right.

The pipeline execution time on the RTX 3070 GPU has been fluctuating in the range of 8 to 10 milliseconds before loading a scene that would utilize all the computational resources of the chip. There is barely any difference in the performance for the cube, cow, white oak, and bunny objects, despite significant differences in the vertex and triangle counts. Erato object utilizes the whole GPU, raising the computational time to an average of almost 15 milliseconds. The complex hairball object with almost 3 million triangles does not fit into the render target of 60 frames per second by a considerable margin. The execution of the pipeline is reasonably consistent and does not fluctuate.

MX250	Cube	Cow	White Oak
Average (μs)	98,358	99,507	176,276
Std (μs)	69,443	39,483	35,160
Relative (%)	70.60	39.68	19.95
	Bunny	Erato	Hairball
Average (μs)	409,886	343,405	—
Std (μs)	77,765	57,817	—
Relative (%)	18.97	16.84	—

Table 6.3. Measurements of total pipeline execution times of 360 degrees camera rotation benchmark on MX250 GPU. Measurements for the hairball object are not provided, the pipeline raised errors when allocating a buffer for vertices.

With the MX250 mobile chip inside a notebook, it is a different story. The GPU failed to achieve consistent execution of the pipeline, fluctuating considerably due to the almost instant power throttling of the GPU, even with the notebook plugged in and set for the best performance. The standard deviation reaches at least 15 percent of the average execution time in every scenario. All the scenes failed to meet the 60 frames per second target. Moreover, the hairball model was not loaded and benchmarked since the GPU failed to allocate a large enough vertex buffer to accommodate all the positions of 1.47 million vertices. The power throttling is observed in the graph of the results of any of the benchmark scenes. Substantial spikes are visible in each of the stages utilizing the GPU.

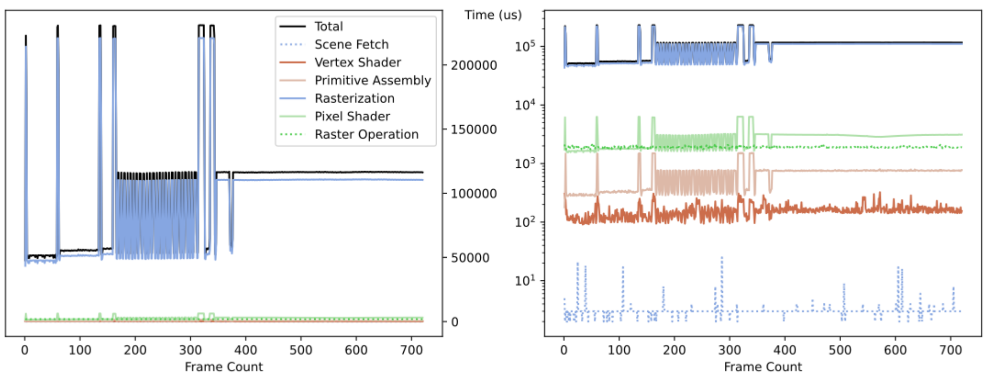


Figure 6.4. Cow 360 degrees camera rotation benchmark results on the MX250 GPU. The massive spikes are visible on the left graph in the rasterization stage. The logarithmic scale of the y-axis on the right graph reveals spikes on every pipeline stage running on the GPU.

6.3.2 Camera Zoom

The performance of the pipeline executing the camera zoom follows the same scheme as the previous benchmark. The rasterization's domination of the execution times is even more underlined when the camera zooms out enough. Hence, the whole object falls into very few bins, with few threads processing all the input triangles. Such behavior highlights the most significant bottleneck of the implementation.

Although the average total execution times did not increase, standard deviations did by a lot. Most of the geometry was view frustum culled at the beginning of the camera zoom benchmark due to the camera being very close to the origin, thus reducing the execution time of the rasterization stage, which stays constant until reaching a distance when the triangles appear again. The appearance is visible on the graphs of the benchmark results at around 150 frame count when the otherwise constant total and rasterization execution times start to grow slowly. The execution time rises quickly once the object shrinks enough to few bins, thus averaging the execution time to the measured values of the 360 degrees camera rotation benchmark. The results show the rasterization executing faster when most triangles are culled, highlighting the benefits of the back-face culling performed in the primitive assembly stage.

RTX 3070	Cube	Cow	White Oak
Average (μs)	8,334	8,378	9,708
Std (μs)	325	389	1,323
Relative (%)	3.90	4.64	13.63
	Bunny	Erato	Hairball
Average (μs)	12,092	22,292	130,113
Std (μs)	4,289	14,618	42,322
Relative (%)	35.47	65.57	32.53

Table 6.4. Measurements of total pipeline execution times of camera zoom benchmark on RTX 3070 GPU.

MX250	Cube	Cow	White Oak
Average (μs)	67,408	115,974	200,901
Std (μs)	36,808	51,697	79,286
Relative (%)	54.61	44.58	39,47
	Bunny	Erato	Hairball
Average (μs)	419,894	316,350	—
Std (μs)	93,613	133,330	—
Relative (%)	22.29	42.15	—

Table 6.5. Measurements of total pipeline execution times of camera zoom benchmark on MX250 GPU.

The hypothesis of the MX250 GPU being theoretically 20 times slower than the powerful RTX 3070 almost holds for both benchmarks. In some computationally lighter scenarios, the performance of the MX250 stays within a multiple of 10 of the RTX 3070. However, it falls toward the performance multiplier 20 for the more complex scenes. The implemented pipeline is practically unusable on the MX250 chip, failing to reach the 60 frames per second target even on the geometrically least complex cube scene with just eight vertices and twelve triangles. Although the performance is sufficient when the pipeline is executed on a cooled idling system, the GPU gets power throttled significantly after a few minutes of the runtime. Such behavior is illustrated in the following figure, depicting two consecutive runs of a buddha scene with 1.08 mil-

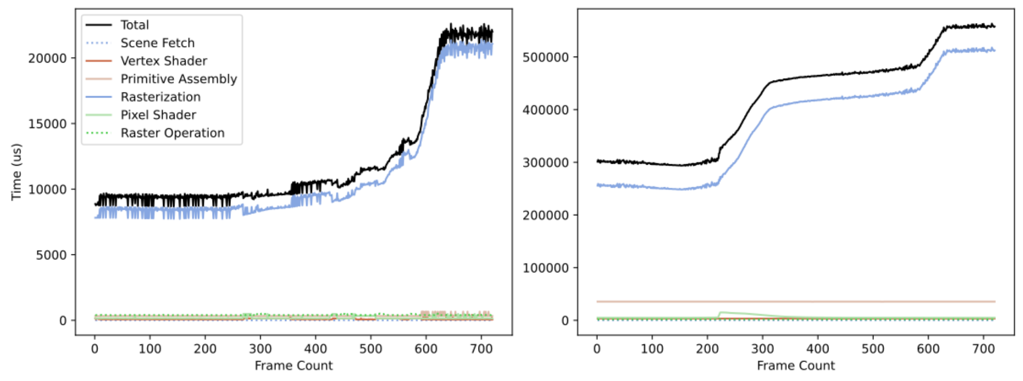


Figure 6.5. Bunny camera zoom benchmark results on the RTX 3070 on the left and MX250 on the right. The rasterization execution time stays constant until around the 200th frame where the bunny geometry stops being view-frustum culled. The time to compute then raises proportionally with the shrinking of the object on the screen.

lion triangles on MX250. The RTX 3070 GPU meets the target of 60 frames per second up to the scene complexity of around half a million triangles.

6.3.3 Resolution

The previous benchmarks confirmed that the rasterization stage of the graphics pipeline would be the bottleneck. They used a resolution of 1280×1024 pixels, 10×8 bins, and 80×64 tiles per window screen, which is the most performant. The following experimentation with the resolution values presents the performance of various configurations, focusing solely on the rasterization stage. All three kernel executions are timed individually. The constraint of the 32 multiple is imposed so that each kernel can execute efficiently in warps of 32 threads according to the CUDA execution model, with each thread being assigned a workload. Resolutions can be changed easily by modifying the corresponding definitions in the code. Each resolution was evaluated on the 360 degrees camera rotation benchmark on a scene with a dragon mesh.

1,280×1,024 pixels	80×64 tiles 20×16 bins	80×64 tiles 10×8 bins	80×64 tiles 5×4 bins
Binning (μs)	69,969	16,678	9,729
Tiling (μs)	1,018	4,828	25,126
Rasterization (μs)	2,583	2,165	2,106
	160×128 tiles 20×16 bins	160×128 tiles 10×8 bins	160×128 tiles 5×4 bins
Binning (μs)	77,212	20,774	13,310
Tiling (μs)	4,016	19,224	107,026
Rasterization (μs)	1,095	985	959

Table 6.6. Measurements of different 1,280×1,024 pixel configurations. The presented values are the average execution times of the three kernels launched in the rasterization stage measured from the 360 degrees camera rotation benchmark.

Generally, increasing the number of containers to sort in any kernel, be it sorting into bins, tiles, or rasterizing the fragments, results in a longer execution time. How-

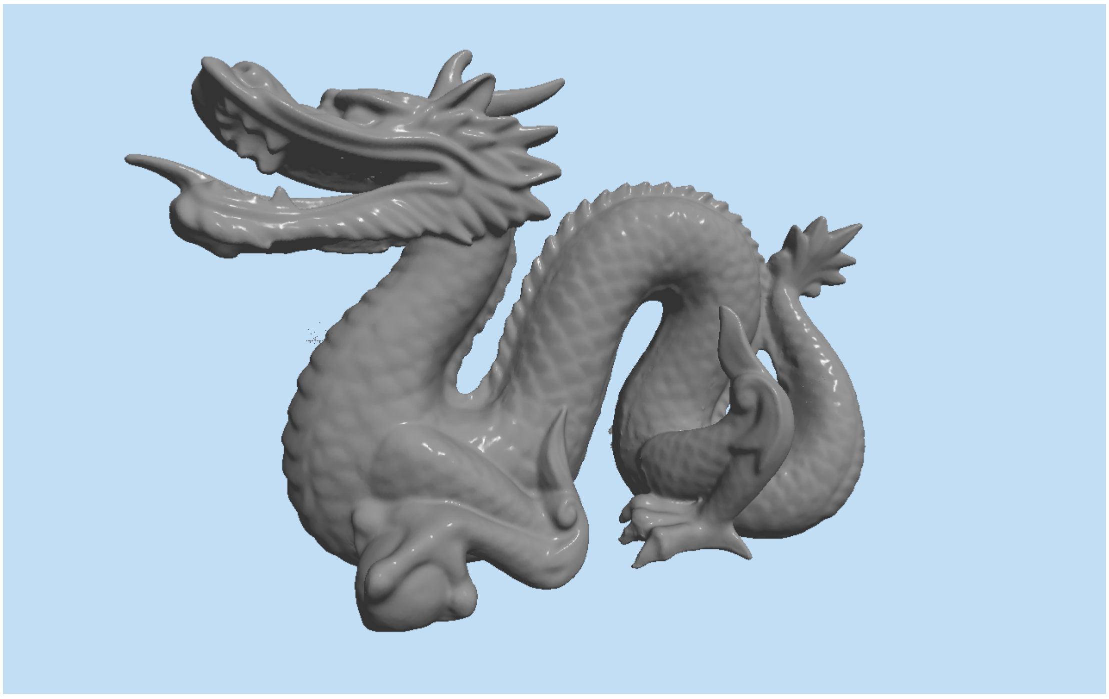


Figure 6.6. Dragon scene with 425,545 vertices and 871,306 triangles.

ever, since the increase of containers sorts the triangles into smaller parts of the screen, the subsequent kernels are processed faster. For example, splitting the screen into 20×16 bins, with each bin holding 4×4 tiles, the execution of the binning kernel takes 70 milliseconds on average, with the tiling kernel computing for a single millisecond. In contrast, reducing the number of bins to 5×4 shortens the execution time of the binning kernel to 10 milliseconds but extends the time of the tiling kernel, which then must sort into more tiles of each bin. The compromise between the two naturally balances the kernel execution times.

Increasing the number of tiles by reducing the tile size of 16×16 fragments per tile to 8x8 fragments halves the execution time of the third rasterization kernel. However, the increase in the computation times of the other two kernels makes this choice hardly justifiable. On the other hand, increasing the size of tiles to 32×32 fragments per tile and reducing the number of tiles from 80×64 to 40×32 offers promising performance, surpassing the one used in the benchmarks for small models. Unfortunately, it errors on the more complex models such as the hairball model since it needs to allocate more enormous queues for triangle sorting to prevent bin and tile buffer overflows, and such allocation request fails. The same principles apply when lowering the screen resolution, with the pipeline running faster.

6.3.4 Reference

The last benchmark was conducted on a buddha to compare the thesis implementation against the references of FreePipe [1], CUDAraster [2], and cuRE [3]. The reference results are published together with the source code of the cuRE implementation. The thesis implementation was evaluated on the GTX 1080 GPU on a scene with a resolution 1,280×1,024 pixels and compared to the measured values of the same graphics card of a 1,024×768 pixels buddha scene.

640×512 pixels	40×32 tiles 10×8 bins	40×32 tiles 5×4 bins
Binning (μs)	11,218	4,075
Tiling (μs)	1,291	7,500
Rasterization (μs)	3,840	5,485
	80×64 tiles 10×8 bins	80×64 tiles 5×4 bins
Binning (μs)	13,190	5,953
Tiling (μs)	5,567	30,571
Rasterization (μs)	1,939	1,930

Table 6.7. Measurements of different 640×512 pixel configurations. The presented values are the average execution times of the three kernels launched in the rasterization stage measured from the 360 degrees camera rotation benchmark.

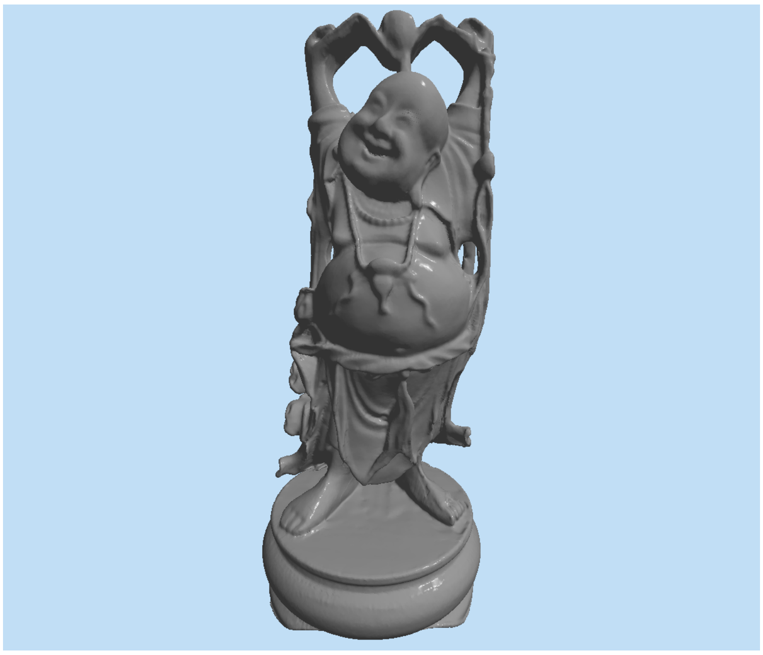


Figure 6.7. Buddha scene with 543,524 vertices and 1,087,474 triangles.

	thesis	FreePipe	cuRE	CUDAraster
Average (ms)	197.3	0.8	7.8	4.2

Table 6.8. Performance comparison of reference implementations.

The buddha scene was evaluated by averaging the execution speed of 300 pipeline passes of the thesis implementation on the GTX 1080 GPU. FreePipe and cuRE were also benchmarked on the same graphics card, while the cuRE was only evaluated on the less powerful GTX 780 Ti but still running faster. The FreePipe implementation benefits from many small triangles of the buddha scene being the fastest implementation, performing way worse in other scenarios. For example, the sponza scene took

78 milliseconds to render on GTX 1080 with FreePipe [3], while the thesis implementation averages 110 milliseconds on a throttled and way less powerful mobile MX250 GPU. However, FreePipe is the only implementation the thesis can even remotely outpace. Compared to the whopping speeds of cuRE and mainly CUDAraster, the thesis implementation runs significantly slower. But it runs.

6.4 Bottlenecks

As the benchmark results clearly showed, the primary critical bottleneck of the graphics pipeline is the rasterization stage and, specifically, the sorting of triangles into bins and tiles. All the other stages run independently in a data-parallel fashion, but the rasterization stage must deal with the depth of fragments, comparing multiple of them. Two solutions to solving the problem of rasterization and depth testing are either taking a triangle and iterating through its pixels, utilizing a screen-wide depth buffer as implemented in the GPU hardware, or taking a pixel and iterating through its queue of triangles omitting the global depth buffer, but introducing the need to sort the triangles, as implemented in the thesis.

The thesis implementation started with experiments with the first approach. However, it soon stumbled upon the problem of efficiently implementing a software screen-wide depth buffer. Taking a triangle and splitting it into multiple fixed-sized bins of pixels processed by a group of CUDA threads is arguably a more elegant and effective solution for rasterization. However, the rasterized fragments must be compared and discarded by their depth and swapped atomically in the depth buffer. The fragment is a structure holding various data, it is not a single integer value, and thus there are no atomic CUDA operations that could swap a fragment for another. Naturally, the swap could be serialized to avoid data races, but then there is no point in running the rasterization on the massively parallel GPU. Another solution is not to care about the data races. However, the rendered output would not be deterministic, and there is a risk of flickering, as seen in the implementation when bin or tile overflow happens, especially when rendering 60 times a second. The fixed-function rasterization units on the graphics chip provide the pipeline with hardware accelerated depth buffer that can handle the depth testing efficiently in parallel. However, the depth buffer is not accessible through the GPGPU techniques.

Thus, the first approach was discarded, and the second was implemented, unfortunately, with its own set of bottlenecks. The pivotal issue of splitting the screen into bins and tiles and sorting the triangles is the non-existent uniformity of the input. The idea was to implement a binning algorithm that would dynamically adapt to the data being rendered, leaving larger bins for the screen regions with fewer triangles and smaller bins for dense areas. However, such an orchestration would most probably demand too much computation time and too complex logic to be worth it. Additionally, increasing the number of bins to sort to does not automatically translate to increased speed, as the resolution experimentation showed. The other solution to a dynamic adaptation to the scene requirements would be to shrink or enlarge the bin and tiles queues the triangles are sorted in. However, this would require reallocating the queue buffers at the runtime, with memory management functions being expensive. A monolithic buffer could be allocated and then split on the fly through indexing and dangling pointers instead. However, this would require computing the queues' sizes before sorting. Otherwise, the enlarging of queues could overwrite other queues.

In the end, the number of bins and tiles is fixed, and all bins and tiles sizes are uniform. Surprisingly, such a naive and straightforward solution is implemented in the fastest software-based renderer CUDAraster. There may be no better solution. The real issue arises when a complex object ends up in a single bin, with all the other bins and its threads being left out without a workload, wasting computational resources and reducing performance. Such behavior is illustrated with the camera zoom benchmark of the thesis implementation. Degenerate culling and the far clipping plane may help, but more is needed. The design of the constant screen splitting is also inappropriate for the production-class rendering pipeline used, for example, in a game engine. The main character with detailed geometry stands in the middle of the screen, with the rest being less dense. The same scenario applies to the thesis implementation since it only supports loading a single mesh that resides at the origin around which a camera orbits.

Apart from being memory hungry and spatially inefficient, the triangle sorting technique of the implementation is also slow. The queue of each bin and tile is filled by atomically incrementing a queue counter, which returns an index to the buffer that is guaranteed to be held only by a single thread. Because the atomic variable is frequently accessed, it is saved in the shared memory to improve writes and reads drastically. However, the memory is only shared between threads inside the same block, and the maximum number of threads in a block is currently 1,024. This means that a bin with a mesh of a few million triangles is only processed with 1,024 threads, each sorting a few hundred, if not thousands, of triangles, making the execution of the sorting extremely slow. CUDAraster processes the input triangles in streams of batches, creating and merging multiple queue buffers of a single bin and employing other low-level knowledgable optimizations. The NVIDIA researchers surely know their hardware the best.

Apart from the colossal rasterization stage bottleneck, there are only missing pipeline features, such as an actual view-frustum clipping, texturing, or support for multiple meshes, which could be implemented in the future. However, the priority should be the rasterization stage.

Chapter 7
Conclusion

The thesis successfully implemented the CUDA-based software renderer. By harnessing the computational capabilities of NVIDIA GPUs, the entirely software-based graphics pipeline achieved suitable performance for most scenarios on a desktop-class GPU. The implemented software renderer efficiently utilized CUDA cores, effectively parallelizing the rendering pipeline and distributing the workload across multiple threads. Multiple thorough benchmarks were conducted, focusing mainly on the rasterization stage, which is the implementation bottleneck. The various solutions with their respective trade-offs were discussed. Apart from the implementation, the thesis presents the reader with a description of the architecture of modern graphics processing units, introducing their history and putting many design considerations into context.

References

[1] Liu, Fang, Meng-Cheng Huang, Xue-Hui Liu, and En-Hua Wu. FreePipe: A Programmable Parallel Rendering Architecture for Efficient Multi-Fragment Effects. *I3D '10: Proceedings of the ACM SIGGRAPH symposium on Interactive 3D Graphics and Games*. New York, NY: Association for Computing Machinery, February, 2010, pp. 75–82. Available from DOI 10.1145/1730804.1730817.

[2] Laine, Samuli, and Tero Karras. High-Performance Software Rasterization on GPUs. *HPG '11: Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*. New York, NY: Association for Computing Machinery, August, 2011, pp. 79–88. Available from DOI 10.1145/2018323.2018337.

[3] Kenzel, Michael, Bernhard Kerbl, Dieter Schmalstieg, and Markus Steinberger. A High-Performance Software Graphics Pipeline Architecture for the GPU. *ACM Transactions on Graphics*. New York, NY: Association for Computing Machinery, August, 2018, Vol. 37, No. 4, pp. 1-15. ISSN 0730-0301. Available from DOI 10.1145/3197517.3201374.

[4] Fatahalian, Kayvon, and Mike Houston. A Closer Look at GPUs. *Communications of the ACM*. New York, NY: Association for Computing Machinery, October, 2008, Vol. 51, No. 10, pp. 50-57. ISSN 0001-0782. Available from DOI 10.1145/1400181.1400197.

[5] Patterson, David A., and John L. Hennessy. *Computer Organization and Design: the Hardware/Software Interface*. 3rd ed. San Francisco, CA: Morgan Kaufmann Publishers, 2005. ISBN 1-55860-604-1.

[6] Šimeček, Ivan, and Jaroslav Sloup. *Obecné výpočty na grafických procesorech*. 2nd ed. Prague: Czech Technical University in Prague, 2017. ISBN 978-80-01-06094-0.

[7] Barlas, Gerassimos. *Multicore and GPU Programming: an Integrated Approach*. 2nd ed. Cambridge, MA: Elsevier, 2023. ISBN 978-0-12-814120-5.

[8] Kirk, David B., and Wen-mei W. Hwu. *Programming Massively Parallel Processors: a Hands-On Approach*. 3rd ed. Cambridge, MA: Elsevier, 2017. ISBN 978-0-12-811986-0.

[9] Žára, Jiří, Bedřich Beneš, Jiří Sochor, and Petr Felkel. *Moderní počítačová grafika*. 2nd ed. Brno: Computer Press, 2005. ISBN 80-251-0454-0. Available from `https://dcgi.fel.cvut.cz/ModerniPocitacovaGrafika/`.

[10] Sloup, Jaroslav. *Algoritmy počítačové grafiky*. Available from `https://cent.felk.cvut.cz/courses/APG/skripta/index.html`.

[11] NVIDIA Corporation. *NVIDIA Technical Blog: CUDA Refresher*. Available from `https://developer.nvidia.com/blog/tag/cuda-refresher/`.

[12] Khronos Group. *Rendering Pipeline Overview*. Available from `https://www.khronos.org/opengl/wiki/Rendering_Pipeline_Overview`.

[13] NVIDIA Corporation. Life of a Triangle - NVIDIA's Logical Pipeline. [online]. NVIDIA Corporation, March 16, 2015. [cit. 2023/5/21]. Available from `https://developer.nvidia.com/content/life-triangle-nvidias-logical-pipeline`.

[14] Giesen, Fabian. *A trip through the Graphics Pipeline*. Available from `https://fgiesen.wordpress.com/2011/07/09/a-trip-through-the-graphics-pipeline-2011-index/`.

[15] Khronos Group. *Vulkan*. Available from `https://www.vulkan.org/`.

[16] Khronos Group. *OpenGL*. Available from `https://www.opengl.org/`.

[17] NVIDIA Corporation. *NVIDIA CUDA*. Available from `https://docs.nvidia.com/cuda/doc/index.html`.

[18] Khronos Group. *OpenCL*. Available from `https://www.khronos.org/opencl/`.

[19] NVIDIA Corporation. *GTC 2023 Keynote with NVIDIA CEO Jensen Huang*. Available from `https://youtu.be/DiGB5uAYKAg`.

[20] Taiwan Semiconductor Manufacturing Company. *3nm Technology*. Available from `https://www.tsmc.com/english/dedicatedFoundry/technology/logic/l_3nm`.

[21] Advanced Micro Devices. *AMD Ryzen 9 5950X Desktop Processors*. Available from `https://www.amd.com/en/products/cpu/amd-ryzen-9-5950x`.

[22] TechPowerUp. *NVIDIA GeForce RTX 3070*. Available from `https://www.techpowerup.com/gpu-specs/geforce-rtx-3070.c3674`.

[23] TechPowerUp. *NVIDIA GeForce MX250*. Available from `https://www.techpowerup.com/gpu-specs/geforce-mx250.c3353`.

[24] NVIDIA Corporation. *NVIDIA's Next Generation CUDA Compute Architecture: Fermi*. Available from `https://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf`.

[25] NVIDIA Corporation. *NVIDIA Volta Architecture*. Available from `https://www.nvidia.com/en-gb/data-center/volta-gpu-architecture/`.

[26] Kilgariff, Emmett, Henry Moreton, Nick Stam, and Brandon Bell. NVIDIA Turing Architecture In-Depth. [online]. NVIDIA Corporation, September 14, 2018. [cit. 2023/8/20]. Available from `https://developer.nvidia.com/blog/nvidia-turing-architecture-in-depth/`.

[27] NVIDIA Corporation. *NVIDIA Ada Lovelace Architecture*. Available from `https://www.nvidia.com/en-us/geforce/ada-lovelace-architecture/`.

[28] NVIDIA Corporation. *NVIDIA RTX Platform*. Available from `https://developer.nvidia.com/rtx`.

[29] NVIDIA Corporation. *RTX Global Illumination*. Available from `https://developer.nvidia.com/rtx/ray-tracing/rtxgi`.

[30] NVIDIA Corporation. *NVIDIA OptiX AI-Accelerated Denoiser*. Available from `https://developer.nvidia.com/optix-denoiser`.

[31] NVIDIA Corporation. *NVIDIA DLSS*. Available from `https://developer.nvidia.com/rtx/dlss`.

[32] NVIDIA Corporation. *NVIDIA CUDA-X: GPU-Accelerated Libraries*. Available from `https://developer.nvidia.com/gpu-accelerated-libraries`.

[33] OpenACC Organization. *OpenACC*. Available from `https://www.openacc.org/`.

[34] OpenMP Architecture Review Boards. *OpenMP*. Available from `https://www.openmp.org/`.

[35] Blender Foundation. *Blender*. Available from `https://www.blender.org/`.

[36] McGuire, Morgan. *Computer Graphics Archive*. Available from `https://casual-effects.com/data/`.

[37] Windows Corportaion. *Programming reference for the Win32 API*. Available from `https://learn.microsoft.com/en-us/windows/win32/api/`.

[38] NVIDIA Corporation. *CUDA C++ Programming Guide*. Available from `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`.

[39] Aslantas, Recep. *Highly Optimized Graphics Math for C*. Available from `https://github.com/recp/cglm`.

[40] G-Truc Creation. *OpenGL Mathematics*. Available from `https://github.com/g-truc/glm`.

[41] FileFormat.info. *Wavefront OBJ File Format Summary*. Available from `https://www.fileformat.info/format/wavefrontobj/egff.htm`.

[42] Wolfram Research. *Wolfram MathWorld: Barycentric Coordinates*. Available from `https://mathworld.wolfram.com/BarycentricCoordinates.html`.

[43] Möller, Tomas, and Ben Trumbore. *Fast, minimum storage ray-triangle intersection*. Available from `http://www.graphics.cornell.edu/pubs/1997/MT97.html`.

Appendix A
User Manual

A.1 Compilation

A custom build script was created for the compilation of the thesis. Make sure to have Nvidia's nvcc compiler with Microsoft's MSVC compiler installed on the system. There are two build configurations, the reader is advised to run the `debug`, with the `release` left for benchmarking. Compile with the following command:

```
.\build.ps1 <target>
```

Not specifying the target prints help message. Verbose output is available by running the following:

```
.\build.ps1 <target> -verbose $true
```

Clean with:

```
.\build.ps1 clean
```

Build the `convert.cu` file and use it as an utility to convert .obj files to a suitable application format:

```
nvcc src\convert.cu
```

Running the utility without arguments prints help:

```
.\convert.exe
```

The pipeline executable is located in `bin\<target>\`. To run it:

```
.\bin\<target>\CUDA-pipeline.exe
```

There are multiple arguments the user can set:

```
-align // Aligns terminal and window next to each other
-layout // Flips Y and Z for CZ keyboard layouts
-verbose // Enables verbose output
-model= // Sets model to load, the model must be in obj folder
-bin= // Sets size of bin queue buffers
-tile= // Sets size of tile queue buffers
-degenerate= // Sets degenerate factor
```

A.2 Controls

Following are the controls of the application:

```
Backspace // Verbose output in the terminal
Escape // Quit

1 // Draw triangles
Shift + 1 // Draw lines
Ctrl + Shift + 1 // Draw points
2 // Smooth shading
Shift + 2 // Flat shading
3 // Ambient component
Shift + 3 // Diffuse component
Ctrl + Shift + 3 // Specular component
4 // Blinn-Phong
Shift + 4 // Camera reflector
5 // Normal vectors
Shift + 5 // Depth buffer
M // Toggle back-face culling
Shift + M // Toggle view-frustum culling

// Following camera controls can be altered with Shift and Ctrl + Shift
Q // Tilt down
E // Tilt up
W // Zoom in
S // Zoom out
A // Rotate left
D // Rotate right

// For the following, Ctrl + Shift modifier resets to default
// and Shift does the opposite change
Z // Decrease FOV
X // Move near clip plane
C // Move far clip plane
R, T, Y // Move model
F, G, H // Scale model
V, B, N // Rotate model
U, I, O // Move light
```