UNIVERSITY OF THESSALY

SCHOOL OF ENGINEERING

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

REAL-TIME ACCELERATED RAY TRACING IN 3D GRAPHICS USING CUDA

Diploma Thesis

Paschalis Choropanitis

Panayiotis Yiannoukkos

Supervisor: Tsalapata Hariklia

Volos 2023

UNIVERSITY OF THESSALY

SCHOOL OF ENGINEERING

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

REAL-TIME ACCELERATED RAY TRACING IN 3D GRAPHICS USING CUDA

Diploma Thesis

Paschalis Choropanitis

Panayiotis Yiannoukkos

Supervisor: Tsalapata Hariklia

Volos 2023

ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ

ΠΟΛΥΤΕΧΝΙΚΗ ΣΧΟΛΗ

ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΕΠΙΤΑΧΥΝΟΜΕΝΗ ΙΧΝΗΛΑΤΗΣΗ ΑΚΤΙΝΑΣ ΣΕ ΠΡΑΓΜΑΤΙΚΟ ΧΡΟΝΟ ΣΕ 3Δ ΓΡΑΦΙΚΑ ΧΡΗΣΙΜΟΠΟΙΩΝΤΑΣ CUDA

Διπλωματική Εργασία

Πασχάλης Χωροπανίτης

Παναγιώτης Γιαννούκκος

Επιβλέπων/πουσα: Τσαλαπάτα Χαρίκλεια

Βόλος 2023

Approved by the Examination Committee:

Supervisor **Tsalapata Hariklia**

Laboratory Teaching Staff, Department of Electrical and Computer Engineering, University of Thessaly

Member **Christos Antonopoulos**

Associate Professor, Department of Electrical and Computer Engineering, University of Thessaly

Member **George Thanos**

Laboratory Teaching Staff, Department of Electrical and Computer Engineering, University of Thessaly

DISCLAIMER ON ACADEMIC ETHICS AND INTELLECTUAL PROPERTY RIGHTS

«Being fully aware of the implications of copyright laws, I expressly state that this diploma thesis, as well as the electronic files and source codes developed or modified in the course of this thesis, are solely the product of my personal work and do not infringe any rights of intellectual property, personality and personal data of third parties, do not contain work / contributions of third parties for which the permission of the authors / beneficiaries is required and are not a product of partial or complete plagiarism, while the sources used are limited to the bibliographic references only and meet the rules of scientific citing. The points where I have used ideas, text, files and / or sources of other authors are clearly mentioned in the text with the appropriate citation and the relevant complete reference is included in the bibliographic references section. I also declare that the results of the work have not been used to obtain another degree. I fully, individually and personally undertake all legal and administrative consequences that may arise in the event that it is proven, in the course of time, that this thesis or part of it does not belong to me because it is a product of plagiarism».

The declarants

Paschalis Choropanitis and Panayiotis Yiannoukkos

Diploma Thesis

REAL-TIME ACCELERATED RAY TRACING IN 3D GRAPHICS USING CUDA

Paschalis Choropanitis

Panayiotis Yiannoukkos

Abstract

The presented thesis discusses an in-depth study and implementation of a real-time ray tracer that utilizes the computational power of Graphics Processing Units (GPUs) through CUDA [1]. Traditional CPU-based ray tracers often face challenges in meeting the high computational demands of ray tracing, resulting in suboptimal performance. To overcome these challenges, a GPU-based approach is adopted, leveraging the massive parallelism of GPUs to accelerate ray tracing tasks. The thesis begins with an exploration of the fundamentals of ray tracing, taking inspiration from Peter Shirley's ”Ray Tracing in One Weekend” 3.1 as a guide. The focus then shifts to the Compute Unified Device Architecture (CUDA) for GPU programming. The approach encompasses various optimization techniques, including memory management, kernel organization, stack-based traversal, avoidance of virtual functions, and the utilization of bounding volume hierarchies (BVH) [2]. Additionally, a user-friendly interface is developed using ImGui [3], enabling real-time interaction with the system. Rigorous testing demonstrates that the application outperforms the CUDA adaptation of ”Ray Tracing in One Weekend” 3.2 by approximately 800%, highlighting the effectiveness of the optimizations applied. Overall, the thesis showcases the profound impact of GPU programming and CUDA on ray tracing, transforming it from a computationally heavy task to a real-time possibility.

Keywords:

ray tracing, path tracing, CUDA, GPU, real-time, 3d graphics, opengl

Διπλωματική Εργασία

ΕΠΙΤΑΧΥΝΟΜΕΝΗ ΙΧΝΗΛΑΤΗΣΗ ΑΚΤΙΝΑΣ ΣΕ ΠΡΑΓΜΑΤΙΚΟ ΧΡΟΝΟ ΣΕ 3Δ ΓΡΑΦΙΚΑ ΧΡΗΣΙΜΟΠΟΙΩΝΤΑΣ CUDA

Πασχάλης Χωροπανίτης

Παναγιώτης Γιαννούκκος

Περίληψη

Η παρούσα διπλωματική συζητά μια ενδελεχή μελέτη και εφαρμογή ενός real-time ray tracer που χρησιμοποιεί την υπολογιστική ισχύ των Μονάδων Επεξεργασίας Γραφικών (GPUs) μέσω της CUDA [1]. Οι παραδοσιακοί ray tracers βασισμένοι σε CPU συχνά αντιμετωπίζουν προκλήσεις στην ικανοποίηση των υψηλών υπολογιστικών απαιτήσεων του ray tracing, οδηγώντας σε μη βέλτιστη απόδοση. Για να ξεπεραστούν αυτές οι προκλήσεις, υιοθετείται μια προσέγγιση βασισμένη σε GPU, εκμεταλλευόμενη τον μαζικό παραλληλισμό των GPU για την επιτάχυνση των εργασιών ray tracing. Η διπλωματική ξεκινά με μια εξερεύνηση των βασικών στοιχείων του ray tracing, παίρνοντας έμπνευση από τον οδηγό ”Ray Tracing in One Weekend” του Peter Shirley 3.1. Η έμφαση στη συνέχεια μετατίθεται στη CUDA για τον προγραμματισμό σε GPU. Η προσέγγιση περιλαμβάνει διάφορες τεχνικές βελτιστοποίησης, συμπεριλαμβανομένης της διαχείρισης μνήμης, της οργάνωσης του kernel, της διαδικασίας διέλευσης βασισμένης σε στοίβα, της αποφυγής των εικονικών συναρτήσεων και της χρήσης των ιεραρχιών οριακού όγκου (BVH) [2]. Επιπλέον, γίνεται ανάπτυξη μιας, φιλική προς τον χρήστη, διεπαφής με τη χρήση του ImGui [3], επιτρέποντας την real-time επαφή με το σύστημα. Δοκιμές αποδεικνύουν ότι η εφαρμογή υπερτερεί της εφαρμογής ”Accelerated Ray Tracing in One Weekend in CUDA” 3.2 κατά περίπου 800%, υπογραμμίζοντας την αποτελεσματικότητα των εφαρμοσμένων βελτιστοποιήσεων. Συνολικά, η διπλωματική παρουσιάζει την ουσιαστική επίδραση του προγραμματισμού GPU και της CUDA στο ray tracing, μετατρέποντάς το από μια υπολογιστικά βαριά εργασία σε μια real-time δυνατότητα.

Λέξεις Κλειδιά:

ray tracing, path tracing, CUDA, GPU, real-time, 3d graphics, opengl

Table of contents

List of figures

Listings

Abbreviations

e.g.	exempli gratia - for example
i.e.	id est - that is
etc.	et cetera - and other similar things
2D	Two Dimensional
3D	Three Dimensional
GPU	Graphics Processing Unit
CPU	Central Processing Unit
SW	Software
HW	Hardware
RT	Ray Tracing
BVH	Bounding Volume Hierarchy
AABB	Axis-Aligned Bounding Box
API	Application Programming Interface
SM	Streaming Multiprocessors
ALU	Arithmetic Logic Units
CUDA	Compute Unified Device Architecture
SIMT	Single Instruction Multiple Thread
SIMD	Single Instruction Multiple Data
UI	User Interface
UX	User Experience
GUI	Graphical User Interface
VAO	Vertex Array Object
VBO	Vertex Buffer Object
IBO	Index Buffer Object
PBO	Pixel Buffer Object

FBO	Frame Buffer Object
TIR	Total Internal Reflection
RTOW	Ray Tracing in One Weekend

Chapter 1

Introduction

The field of computer graphics has been rapidly advancing over the past few decades, with the development of new techniques for rendering realistic images in real-time. One of the most popular techniques used in rendering is ray tracing, which involves tracing the path of light rays through a virtual scene to create an accurate representation of the scene [7]. However, ray tracing can be a computationally expensive process, especially for complex scenes with many objects and lighting effects.

To address this challenge, Graphics Processing Units (GPUs) have been used to accelerate the rendering process through parallel computing. One such GPU technology is CUDA, which is a parallel computing platform and application programming interface (API) developed by NVIDIA for general purpose cumputing on GPUs [8].

In this thesis, the utilization of CUDA for real-time acceleration of ray tracing in 3D graphics is thoroughly examined. Specifically, the investigation focuses on the performance advantages derived from leveraging CUDA for expediting the ray tracing process, drawing comparisons with conventional CPU-based rendering methodologies.

Overall, the goal of this thesis is to demonstrate the feasibility and potential of using CUDA for real-time accelerated ray tracing in 3D graphics, and to provide insights into how CUDA-based rendering can improve the efficiency and quality of the rendering process.

1.1 Problem Statement

The core issue this thesis seeks to address lies at the heart of modern computer graphics: achieving real-time ray tracing. Ray tracing, though highly renowned for generating high-quality, photorealistic images, poses a significant computational challenge due to its inherent complexity, which prevents its real-time execution on standard hardware.

The mission is to develop a feasible solution that enables real-time ray tracing by leveraging the immense parallel processing power of GPUs. By harnessing NVIDIA’s Compute Unified Device Architecture (CUDA), the objective is to optimize ray tracing to a level where attainment of real-time performance becomes feasible.

This endeavor presents a multitude of technical obstacles. Firstly, there is a need to devise a way to efficiently handle bounding volume hierarchies (BVHs) in a GPU context. BVHs play a vital role in ray tracing, as they drastically reduce the number of required ray-object intersection tests. This issue is addressed by applying a stack-based traversal algorithm adapted to GPU’s parallel environment.

The second challenge involves optimizing memory usage and enhancing performance to ensure the ray tracer operates at the highest efficiency. To this end, several CUDA features such as Coalesced Memory Access are used, which contribute to achieving real-time performance.

Thirdly, the need for an intuitive user interface arises to allow real-time adjustments to the scene and rendering parameters. For this purpose, ImGui, a user-friendly library, is employed to provide an interactive experience for users.

Lastly, in order to validate the approach of this thesis, rigorous testing must be conducted, comparing the application’s performance against both traditional CPU-based and existing CUDA-based ray tracers. This empirical analysis provides a measure of the improvement this solution offers in the field of real-time ray tracing.

By tackling these challenges, this thesis proposes a feasible, GPU-accelerated, real-time solution to the complex problem of ray tracing, pushing the boundaries of what is currently achievable in computer graphics.

1.1.1 Contribution

This thesis makes several significant contributions to the realm of real-time ray tracing, offering practical solutions to the challenges that were outlined. These include:

1. GPU-accelerated Ray Tracer: An effective ray tracer utilizing CUDA to exploit the GPU’s parallel processing power was developed. This implementation significantly accelerates ray tracing, enabling real-time performance.

2. Adapted BVH Traversal: To efficiently process bounding volume hierarchies (BVHs) on the GPU, a stack-based traversal algorithm suitable for the GPU's parallel architecture was adapted, enhancing the ray tracing efficiency.

3. CUDA Optimizations: Various CUDA features were harnessed to optimize memory usage and performance. These include techniques such as Coalesced Memory Access, contributing to the real-time capability of the ray tracer.

4. User Interface Implementation: ImGui, a user-friendly library, was integrated into the application. This allows users to interact in real-time with the scene and rendering parameters, enhancing user experience and allowing for instant visual feedback.

5. Performance Testing: Extensive testing was conducted to benchmark the application's performance against traditional CPU-based ray tracers and existing CUDA-based ray tracers. This analysis helps to quantify the contributions and validate the approach of this thesis.

These contributions collectively pave the way for achieving real-time ray tracing, thereby marking a significant stride in the field of computer graphics.

1.2 Thesis structure

This thesis is organized into five comprehensive chapters, each serving a distinct purpose in the journey of real-time ray tracing exploration:

1. ***Chapter 1: Introduction*** Introductory chapter setting the stage for the project, outlining the problems this thesis is trying to address, the novel contributions, and the structure of the thesis itself.

2. ***Chapter 2: Background*** delves into the theoretical underpinnings crucial to the study. This includes an overview of ray tracing, computer graphics, GPU architecture, and CUDA programming, setting the knowledge base required to understand the remaining sections.

3. ***Chapter 3: Similar Projects*** examines a selection of existing projects related to the field of study, providing context and highlighting the unique aspects and limitations of these works.

4. ***Chapter 4: Real-Time Path Tracer Implementation*** is the segment where the GPU-accelerated ray tracer is meticulously discussed. The discourse encompasses its design, the explicit CUDA techniques brought into play, as well as the manifestation of the application interfaces tailored to ensure real-time interaction and optimal performance.

5. ***Chapter 5: Experiments, Conclusion, and Future Work*** finalizes the thesis by consolidating all the elements of the research. The chapter discusses testing methodologies that were employed, complemented by a comparison of performance metrics between the new implementation and comparable pre-existing projects. The discourse then shifts to potential enhancements and further developments that could amplify the functionality and performance of the ray tracer.

Through this structure, the objective is to furnish a comprehensive, intelligible, and detailed exposition of the conducted work and its implications within the realm of real-time ray tracing.

Chapter 2

Background

This chapter aims to provide a background on the technologies and concepts that form the foundation for this research on real-time accelerated ray tracing using CUDA. The discussion begins by exploring the architecture and capabilities of Graphics Processing Units (GPUs) and the CUDA platform. Subsequently, the focus shifts to rendering real-time graphics on GPUs using the OpenGL (Open Graphics Library) API [9], as well as the CUDA-OpenGL interoperability [10] that facilitates the seamless integration of CUDA-based computations with OpenGL rendering. Finally, the chapter concludes with an overview of the fundamentals of ray tracing and its evolution over the years.

2.1 Introduction

A GPU is a specialized processor designed for rendering and manipulating images, while a CPU is a general-purpose processor used for a variety of computing tasks. CPUs are designed to handle a wide range of computational tasks sequentially, while GPUs are optimized for parallel processing of large amounts of data simultaneously. This makes GPUs highly effective for tasks that require the processing of massive amounts of data in parallel. GPUs have many more cores than CPUs and can execute thousands of operations simultaneously. This makes them much faster for tasks that can be broken down into many small parallel tasks, such as rendering complex 3D graphics or training deep learning models. Ray tracing, in particular, is a computationally intensive process that requires tracing the path of individual rays of light through a scene to create a realistic image. Because GPUs are optimized for parallel processing, they can perform ray tracing calculations much faster than CPUs, making them the preferred choice for high-performance graphics applications.

2.2 NVIDIA GPU Architecture

The NVIDIA GPU Architecture is a critical component of modern graphics technology. NVIDIA GPUs are designed to provide high-performance parallel computing capabilities that are essential for real-time ray tracing in 3D graphics. This section explores the architecture of NVIDIA GPUs [11].

2.2.1 NVIDIA CUDA Architecture

NVIDIA GPUs are based on a massively parallel architecture that is optimized for data-parallel operations. This architecture is known as the CUDA (Compute Unified Device Architecture) architecture. The CUDA architecture is composed of a hierarchy of processing elements that include threads, blocks, and grids. Such elements can be graphically represented in Figure 2.1.

2.2.2 Threads

A thread is the smallest unit of execution in the CUDA architecture. Threads are executed in parallel on the GPU and are organized into groups called blocks. Each thread is assigned a unique thread ID that is used to identify it during execution.

2.2.3 Blocks and Grids

A block is a group of threads that are executed together on a single Streaming Multiprocessor (SM). Blocks are organized into grids, which are collections of blocks. The number of threads per block and the number of blocks per grid can be customized to optimize performance for specific applications.

2.2.4 Streaming Multiprocessors (SMs)

An SM is a collection of processing elements that include ALUs (Arithmetic Logic Units), registers, and shared memory. Each SM is capable of executing multiple threads in parallel.

The number of SMs in a GPU determines the maximum number of threads that can be executed simultaneously.

2.2.5 Memory Hierarchy

The memory hierarchy of the NVIDIA GPU is designed to provide fast access to data and minimize memory access latency. The memory hierarchy includes several types of memory, including registers, local, shared, global, constant and texture memory.

2.2.6 Registers

Registers in CUDA represent the fastest memory type and are private to each thread. They hold variables that the thread is currently computing. Each CUDA core has a small, dedicated register file, where the number of available registers depends on the GPU architecture and the compute capability. Although their access latency is the lowest, the total number of registers available per block is limited, and excessive usage can limit the number of resident threads per multiprocessor, leading to lower occupancy and potentially decreased performance. However, with careful optimization of register usage, it's possible to greatly increase a CUDA program's efficiency and speed. It's worth noting that the allocation of variables to registers is handled by the compiler, and the developer generally has no direct control over it. The compiler attempts to optimize register usage to maximize parallelism, but in some cases, it might spill over to local memory, which is much slower, when register usage is high.

2.2.7 Local Memory

Local memory in CUDA is a region of device memory private to each thread. It is used for automatic variables that do not fit into the registers. Despite the name, local memory does not have the same low latency and high bandwidth as shared or constant memory. In fact, local memory accesses have similar latency and bandwidth to global memory accesses. The use of local memory can often be a performance bottleneck due to its high latency and low bandwidth, and thus its use is generally avoided when possible. However, it's worth noting that the CUDA compiler and runtime will automatically place large structures or arrays that exceed available register or shared memory space into local memory.

2.2.8 Shared Memory

Shared Memory in CUDA is a user-managed cache that is local to each streaming multiprocessor and shared among all threads within a block. It is one of the fastest memory spaces and can be used for inter-thread communication, data sharing, and result aggregation within a block. However, its size is limited (up to 48 KB or 64 KB per block, depending on the CUDA version and GPU model), and developers have to manage it carefully to prevent race conditions and maximize performance. Shared memory is crucial for achieving high performance in CUDA applications, as it enables coalesced memory access and minimizes expensive global memory transactions.

2.2.9 Global Memory

Global Memory is the largest memory space available in CUDA and can be accessed by all threads as well as the host (CPU). However, it has the highest access latency among all memory spaces. The effective bandwidth of global memory can be significantly improved by ensuring that memory accesses are coalesced, i.e., consecutive threads access consecutive memory locations. Global memory allocations persist for the lifetime of the application and are an excellent place for the input data to the kernels and for storing the results.

2.2.10 Constant Memory

Constant Memory in CUDA is a read-only memory space that resides on the device and is accessible from all threads within a grid. Its primary use is for data that remains unchanged throughout the kernel's execution. Because constant memory is cached on each multiprocessor, the efficiency of memory accesses can be significantly improved, provided the access pattern among threads is the same. This enables simultaneous read access to the same memory address, termed as ”broadcast”, leading to a substantial increase in memory bandwidth. However, if threads access divergent addresses, the cache utilization decreases, and the memory access performance degrades. The total constant memory available on the device is limited (64 KB as of CUDA 10.x), which should be kept in mind while using it.

2.2.11 Texture Memory

Texture memory in CUDA provides a specialized read-only memory space designed for texturing operations in graphics rendering. However, its caching and interpolation capabilities also make it useful for general purpose computing. The texture memory is cached on each streaming multiprocessor, and provides spatial locality caching. That is, if nearby threads read nearby memory locations, the caching efficiency is high, and it can greatly speed up memory access. Texture memory is beneficial when dealing with structured grid data or when memory access patterns exhibit spatial locality. Additionally, texture memory allows for hardware interpolation for floating point values, a feature not available with other memory types.

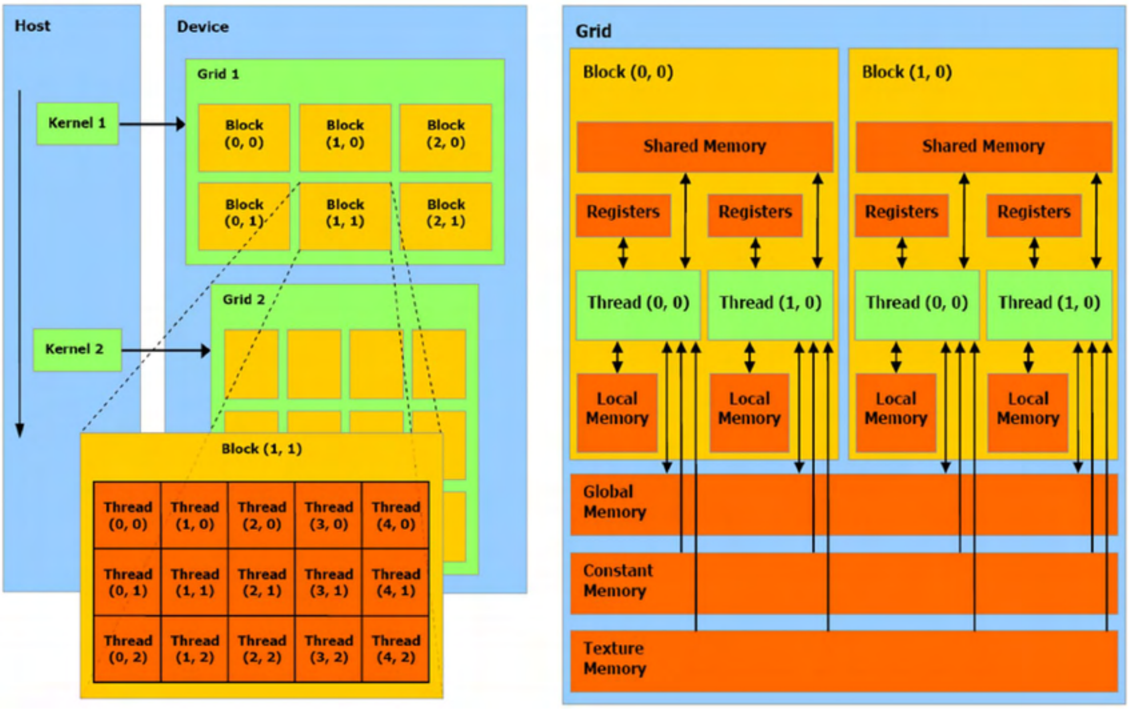


Figure 2.1: Schematization of CUDA architecture [4]

The NVIDIA GPU Architecture is a critical component of modern graphics technology. Its massively parallel architecture, optimized for data-parallel operations, provides high-performance parallel computing capabilities that are essential for real-time ray tracing in 3D graphics. Understanding the architecture of NVIDIA GPUs is essential for developing high-performance applications that utilize the full capabilities of these powerful computing devices.

2.3 CUDA Computing API

This sections explores the CUDA Computing API, which is a programming model and platform for parallel computing on NVIDIA GPUs.

The CUDA Computing API is a high-level programming interface that provides developers with a simple and flexible way to leverage the parallel processing power of NVIDIA GPUs. The API is based on the C programming language and provides extensions that enable developers to write code that can be executed in parallel on the GPU.

2.3.1 Kernel Functions

The core of the CUDA programming model is the kernel function, which is a C function that is executed in parallel on the GPU. Kernel functions are defined using the ***global*** keyword, which tells the CUDA compiler that the function should be executed on the GPU.

The execution of a kernel function is organized into a grid of blocks and threads. The number of blocks and threads in a grid can be customized to optimize performance for specific applications.

2.3.2 Memory Management

Memory management is a critical aspect of parallel programming on the GPU. CUDA provides several types of memory that can be used to store data, including global memory, shared memory, and registers. CUDA provides several memory management functions that can be used to allocate, copy, and free memory on the GPU.

2.3.3 Performance Optimization

Performance optimization is critical for achieving maximum performance when programming on the GPU. CUDA provides several tools and techniques that can be used to optimize performance, including:

- Customizing the size of the grid and the number of threads in each block to optimize the use of GPU resources.

- Using shared memory to reduce memory access latency.

- Minimizing data transfer between the CPU and GPU by using pinned memory.
- Using asynchronous data transfer to overlap data transfer with computation.
- Optimizing memory access patterns to reduce memory conflicts and improve data locality.
- Using CUDA profiler tools to identify performance bottlenecks and optimize code.

2.4 CUDA Programming Model

Both the CPU and GPU are used in the CUDA programming model. The GPU and its memory are referred to as the device in CUDA, while the host refers to the CPU and its memory. In addition to launching kernels, which are operations carried out on the device, code running on the host can manage memory on both the host and the device. These kernels are run in parallel by several GPU threads. With that being said, a typical order of operations when programming in CUDA, is:

1. Allocate any memory resources on both the host and the device.
2. Initialize host data.
3. Copy host data to the device.
4. Execute CUDA kernels.
5. Retrieve the results from the device and copy them back to the host.

Figure 2.2 visualizes the above order of operations.

Listing 2.1 shows a simple CUDA C SAXPY (Single-precision A*X Plus Y) program that abides to the aforementioned order of operations[12].

```
// CUDA Kernel Device code
__global__ void saxpy(const int n, float a, float *x, float *y)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < n)
        y[i] = a*x[i] + y[i];
}

```

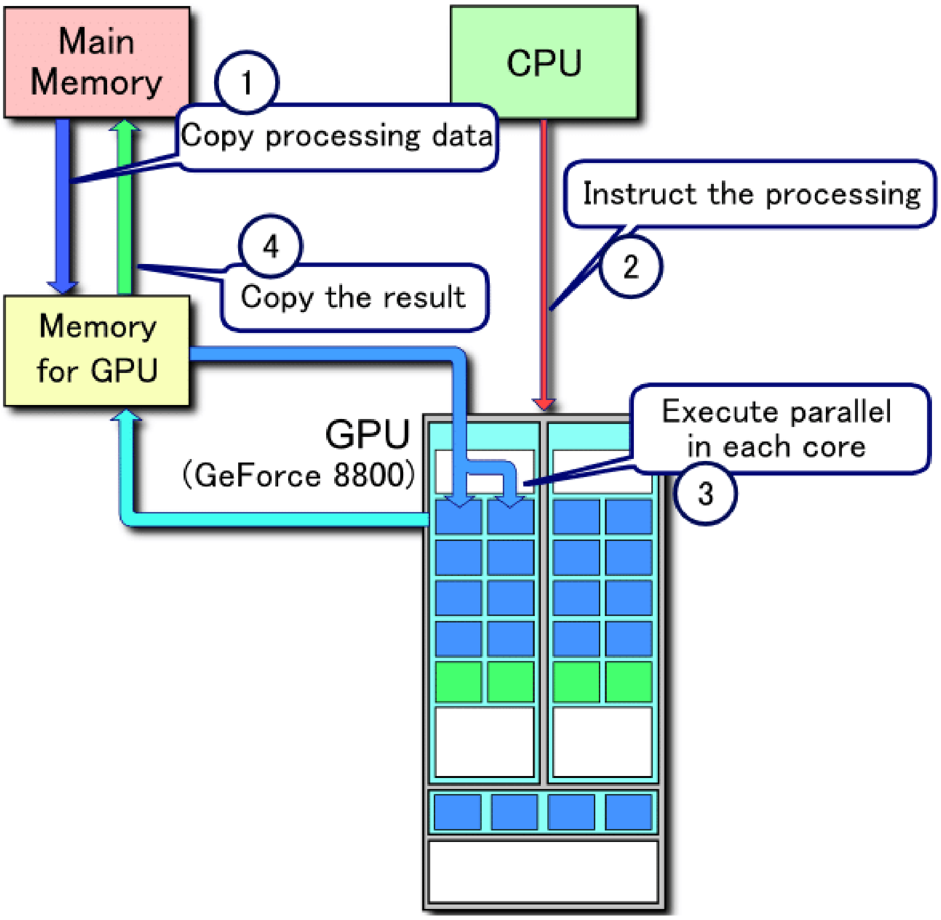


Figure 2.2: Processing flow on CUDA [5]

```
// Host main routine
int main()
{
    const int N = 10000000;
    float *x, *y, *d_x, *d_y;

    // Allocate the host input vectors
    x = (float *)malloc(N * sizeof(float));
    y = (float *)malloc(N * sizeof(float));

    // Allocate the device input vectors
    cudaMalloc(&d_x, N * sizeof(float));
    cudaMalloc(&d_y, N * sizeof(float));

    // Initialize the host input vectors
    for (int i = 0; i < N; i++) {
        x[i] = 1.0f;
        y[i] = 2.0f;
    }

    // Copy the host input vectors in host memory to the device input
    // vectors in device memory
    cudaMemcpy(d_x, x, N * sizeof(float), cudaMemcpyHostToDevice);
    cudaMemcpy(d_y, y, N * sizeof(float), cudaMemcpyHostToDevice);

    // Perform SAXPY on 10M elements
    saxpy<<<(N+255)/256, 256>>>(N, 2.0f, d_x, d_y);

    // Copy the device result vector in device memory to the host result
   vector
    // in host memory
    cudaMemcpy(y, d_y, N * sizeof(float), cudaMemcpyDeviceToHost);

    // Free device global memory
    cudaFree(d_x);
    cudaFree(d_y);
    // Free host memory
    free(x);
    free(y);

    return 0;
}
```

Listing 2.1: Simple CUDA C SAXPY program

Figure 2.3 presents a comparison of the execution time between the CUDA program described above and the equivalent program written in C++ without any GPU acceleration. The tests were conducted on an AMD Ryzen 5 3600 6-Core Processor @3.6GHz CPU and a NVIDIA GeForce RTX 2060 SUPER GPU with 2176 CUDA cores. Upon analysis, it is evident that for a small number of elements, there is no noticeable performance improvement with the CUDA implementation. In fact, there is a performance decrease due to the overhead associated with kernel execution and PCIe data transfer. However, as the number of elements increases, the execution time of the CPU-based version exhibits an exponential increase, whereas the CUDA version either remains constant or even decreases significantly when handling a large number of elements.

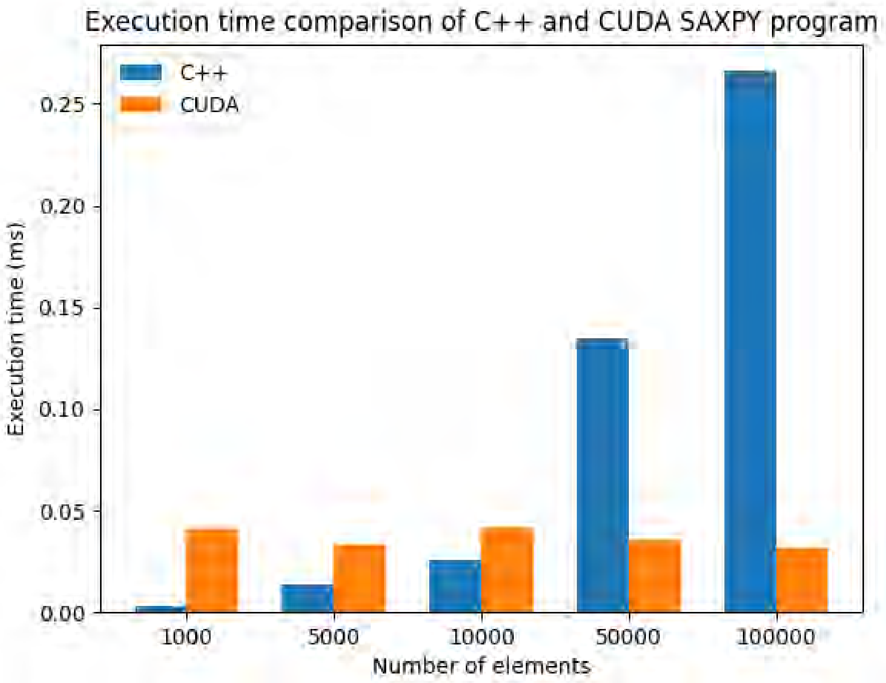


Figure 2.3: Execution time comparison of C++ and CUDA SAXPY program

2.5 CUDA-OpenGL Interoperability

2.5.1 OpenGL

OpenGL (Open Graphics Library) is a cross-platform, open-source API that is widely used for rendering 2D and 3D graphics. Originally developed by Silicon Graphics in the early 1990s, OpenGL has since become an industry standard for graphics rendering in video games, CAD/CAM software, and scientific visualization.

OpenGL provides a set of functions for creating and manipulating 3D objects, lighting, textures, and other visual effects. It also allows for the integration of user input and output, such as mouse clicks and keyboard events. OpenGL is highly configurable and can be optimized for specific hardware and software configurations.

2.5.2 CUDA-OpenGL Interoperability

CUDA-OpenGL interoperability allows developers to combine the power of CUDA with the rendering capabilities of OpenGL. This means that developers can use CUDA to perform calculations on the GPU and then seamlessly transfer the results to OpenGL for rendering.

One of the main benefits of CUDA-OpenGL interoperability is that it enables developers to take advantage of the parallel processing power of the GPU for both computation and rendering. This can result in significant performance improvements, particularly for applications that require real-time rendering of complex 3D scenes.

To enable CUDA-OpenGL interoperability, NVIDIA provides a set of libraries and APIs, including the CUDA Runtime API, the CUDA Driver API, and the OpenGL Interoperability Extension. These libraries and APIs allow developers to share data and resources between CUDA and OpenGL contexts, enabling efficient data transfer and synchronization.

NVIDIA provides a set of samples for CUDA developers which demonstrates features in CUDA Toolkit [13]. One of these samples demonstrates how to achieve interoperability between CUDA and OpenGL using the OpenGL-CUDA graphics interop API [14].

The sample creates a simple texture quad that is rendered using OpenGL. The pattern is stored in a CUDA texture, which is shared with OpenGL using the graphics interop API. The sample first initializes an OpenGL context and creates a window for rendering the scene. It then creates a CUDA context and allocates a CUDA array to hold the texture data. Next, it registers the CUDA array with OpenGL using the graphics interop API and creates an OpenGL texture object to represent the shared texture. The sample then defines a CUDA kernel that computes the texture data and copies it to the CUDA array. The kernel is launched from the CPU and executes on the GPU. Once the computation is complete, the sample renders the texture using OpenGL, which fetches the texture data from the shared CUDA texture. The sample demonstrates how to synchronize the CPU and GPU to ensure that the texture data is fully computed before it is used for rendering.

Overall, the sample provides a practical example of how to achieve CUDA-OpenGL interoperability using the graphics interop API. It shows how to share data and resources between the two platforms and provides insights into how to optimize performance.

In conclusion, CUDA-OpenGL interoperability provides developers with a powerful tool set for creating high-performance 3D graphics applications. By combining the parallel processing power of CUDA with the rendering capabilities of OpenGL, developers can create applications that deliver real-time performance and high-quality visual experiences.

2.6 Ray Tracing: A Brief Overview

Ray Tracing is a rendering technique used in computer graphics to generate images by simulating the physical behavior of light. In contrast to other rendering techniques, such as rasterization, which focus on projecting geometries onto a 2D screen, Ray Tracing is based on tracing the path of individual rays of light as they interact with objects in a 3D scene.

As seen in Figure 2.4, the core idea behind Ray Tracing is to simulate how light travels from a virtual camera, which acts as the eye of the viewer, into the scene. Each ray is cast from the camera and traverses the scene until it hits an object, at which point it bounces off the surface and continues its path, possibly interacting with other objects along the way. Eventually, the ray will either reach a light source, in which case the corresponding pixel in the image is illuminated, or it will escape the scene, in which case the pixel is left in shadow.

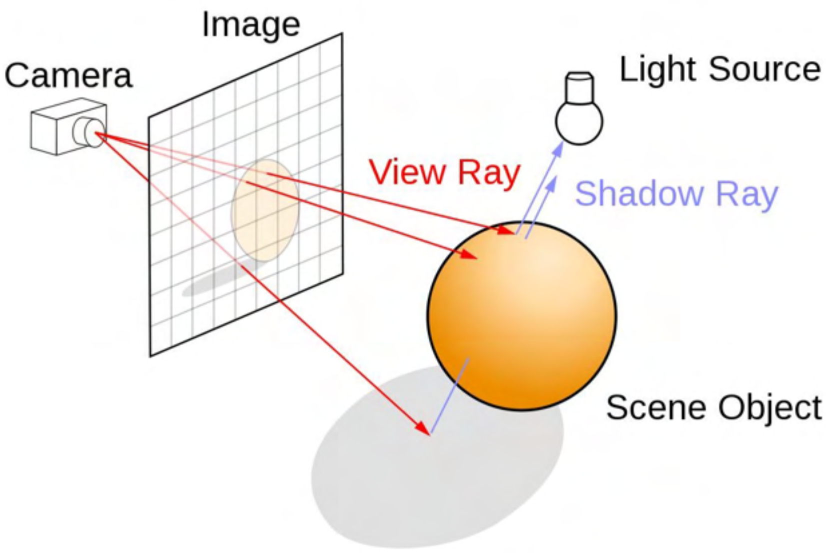


Figure 2.4: Ray Tracing Basics

The Ray Tracing algorithm works by computing the intersection points between the rays and the objects in the scene. For each intersection, the algorithm computes the lighting and shading of the corresponding pixel, taking into account the material properties of the object and the position and intensity of the light sources. The result is a high-quality, photorealistic image that accurately captures the reflections, refractions, shadows, and other optical effects that occur in real-world scenes. A good example can be seen in Figure 2.5.

While Ray Tracing has been used in computer graphics for decades, its popularity has increased significantly in recent years thanks to advances in hardware and software technology. Graphics processing units (GPUs) have become faster and more capable, enabling real-time

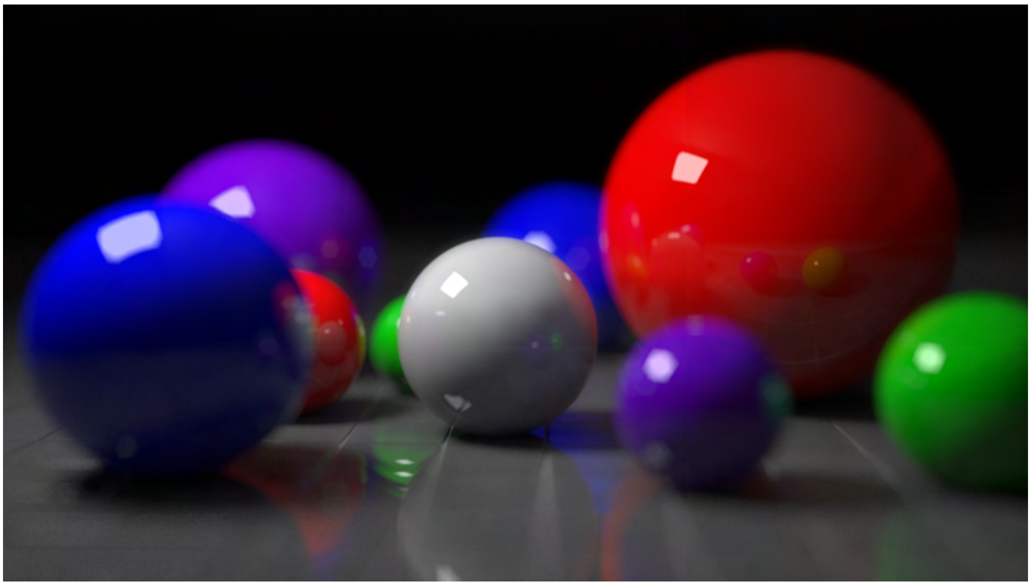


Figure 2.5: Ray Traced Image

Ray Tracing for video games and other interactive applications. In addition, software frameworks such as NVIDIA's CUDA allow developers to leverage the parallel processing power of GPUs to accelerate the Ray Tracing algorithm and achieve even higher levels of performance and realism.

This thesis focuses on exploring the potential of CUDA-accelerated Ray Tracing for real-time 3D graphics applications. By leveraging the power of modern GPUs, the aim is to achieve high levels of performance and interactivity while maintaining photorealistic image quality.

Chapter 3

Similar Projects

3.1 Ray Tracing In One Weekend

The ”Ray Tracing in One Weekend” [15] book by Peter Shirley is a concise, hands-on guide that introduces readers to the fundamentals of ray tracing, a computer graphics technique used to create realistic images through simulating light interactions with objects in a 3D scene. The book is designed for readers with a basic understanding of programming, ideally in C++, and takes them through the process of building a simple ray tracer from scratch over the course of a weekend.

As described in chapter 2, when one mentions the term ”ray tracing,” it can encompass various meanings. In this context, the focus is on developing a brute-force path tracer, which is a relatively general approach. Although the code remains fairly straightforward (allowing the computer to handle the complexity), the resulting images produced by this method are expected to be quite impressive and visually appealing. The approach described in this book is done exclusively on the CPU without any hardware acceleration techniques.

3.1.1 Output an Image

In the second chapter, titled ”Output an Image”, the author teaches the reader how to create a simple image file by outputting color values for each pixel. The author introduces the concept of the Portable PixMap (PPM) image format, a straightforward format that can be easily generated and read by many image editing programs.

The chapter guides the reader through the process of creating a basic PPM file with a set width and height, iterating through each pixel, and assigning color values for the red, green, and blue channels. The resulting output is a gradient image that serves as the foundation for the ray tracing techniques that will be explored in the subsequent chapters of the book.

3.1.2 The vec3 Class

In the third chapter, titled ”The vec3 Class”, the author introduces a 3D vector class, which is an essential data structure for representing points, directions, and colors in the ray tracer. This custom vec3 class will be used throughout the book for various purposes in the ray tracing process.

The author explains the importance of having a vec3 class, as it allows for easier manipulation of 3D coordinates and their respective operations, such as addition, subtraction, scaling, and dot product. Additionally, the vec3 class plays a crucial role in representing colors, where each component (red, green, and blue) corresponds to a specific channel’s intensity.

Shirley provides code samples and explanations for implementing the vec3 class and its associated operations. By the end of this chapter, the reader will have gained a solid understanding of the vec3 class and its applications in a ray tracer, setting the groundwork for the upcoming chapters that delve deeper into ray tracing techniques.

3.1.3 Rays, a Simple Camera, and Background

In the fourth chapter, titled ”Rays, a Simple Camera, and Background”, the reader is introduced to the foundational concepts of rays, camera setup, and background rendering in the context of ray tracing.

The author begins by defining what a ray is: a mathematical construct consisting of an origin point and a direction vector. Rays play a central role in ray tracing, as they simulate light traveling through a scene and interacting with objects. Shirley then demonstrates how to create a simple camera by defining its position, orientation, and field of view. The camera generates rays that pass through each pixel of the image, forming the basis for the ray tracing process.

Furthermore, this chapter explains how to generate a background, which is the image that appears when a ray does not intersect any objects in the scene. Shirley presents a method for creating a simple gradient background that simulates the appearance of the sky.

3.1.4 Adding a Sphere

In the fifth chapter, titled ”Adding a Sphere”, Peter Shirley teaches readers how to render a sphere in their ray tracer. This chapter introduces the concept of ray-object intersection, a crucial aspect of ray tracing that determines whether a ray intersects an object in the scene.

The author begins by explaining the mathematics behind sphere-ray intersection, using the geometric equation of a sphere and the parametric equation of a ray. Shirley then presents an algorithm to calculate if and where a ray intersects a sphere, and how to find the intersection point.

Next, the chapter demonstrates how to incorporate sphere rendering into the existing ray tracer, by iterating through the rays generated by the camera and testing them for intersection with a sphere in the scene. If an intersection is detected, the corresponding pixel in the image is colored based on the sphere’s surface normal, creating a shaded appearance.

3.1.5 Surface Normals and Multiple Objects

In the sixth chapter, titled ”Surface Normals and Multiple Objects”, Peter Shirley delves into the concepts of surface normals and handling multiple objects in a scene. These concepts are essential for creating more complex and visually appealing images in the ray tracer.

First, the author explains the importance of surface normals, which are perpendicular vectors to a surface at a given point. Surface normals are crucial for shading calculations, as they help determine how light interacts with the surface. Shirley demonstrates how to calculate the surface normal of a sphere at the intersection point and how to use this normal to shade the sphere, giving it a more realistic appearance as shown in Figure 3.1.

Next, the chapter introduces the concept of handling multiple objects in a scene. Shirley describes an abstract class called ”hittable” that can represent any object that rays can intersect. The sphere class is then modified to inherit from this ”hittable” class, making it easier to manage multiple spheres in the scene. The author also introduces a ”hittable_list” class that can store and manage multiple hittable objects.

3.1.6 Antialiasing

In the seventh chapter, titled ”Antialiasing”, Peter Shirley introduces the concept of antialiasing, a technique used to reduce the visual artifacts that occur when rendering images at

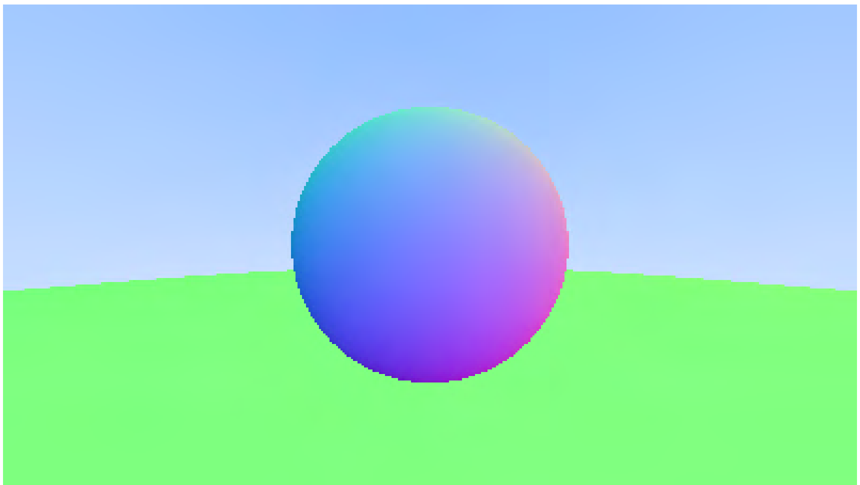


Figure 3.1: Resulting render of normals-colored sphere with ground

a finite resolution. Aliasing can cause jagged edges and pixelation, which degrade the image quality.

The author explains that antialiasing works by sampling multiple points within each pixel and averaging their color values, which smooths out the edges and reduces the appearance of jaggedness. This is achieved by introducing sub-pixel sampling, where multiple rays are cast from different positions within a single pixel, and their resulting colors are averaged to determine the final pixel color. We can see the before and after results in Figure 3.2.

Shirley demonstrates how to implement antialiasing in the ray tracer by modifying the camera and rendering loop. Instead of casting a single ray per pixel, the camera casts multiple rays with slightly varying directions within each pixel. The final color of the pixel is calculated by averaging the colors obtained from these multiple rays.

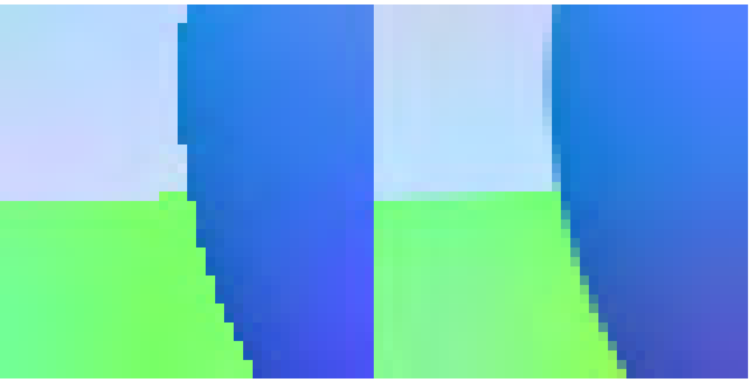


Figure 3.2: Before and after antialiasing

3.1.7 Diffuse Materials

In the eighth chapter, titled ”Diffuse Materials”, Peter Shirley introduces the concept of materials and demonstrates how to implement diffuse shading in the ray tracer. Materials are essential in creating realistic images, as they define how objects in a scene interact with light.

The author begins by explaining the physics of diffuse reflection, which occurs when light scatters uniformly in all directions upon striking a surface. Diffuse materials, such as matte finishes, exhibit this property. To simulate diffuse reflection in the ray tracer, Shirley introduces the concept of scattering, where a ray that hits a diffuse object generates a new, randomly directed ray that continues to trace through the scene. We can see the Lambertian rendering in Figure 3.3.

Shirley demonstrates how to implement a simple diffuse material by modifying the ”hittable” objects and the rendering loop. When a ray intersects an object with a diffuse material, the ray tracer generates a new scattered ray, which is then traced recursively to calculate the final color contribution from the object.

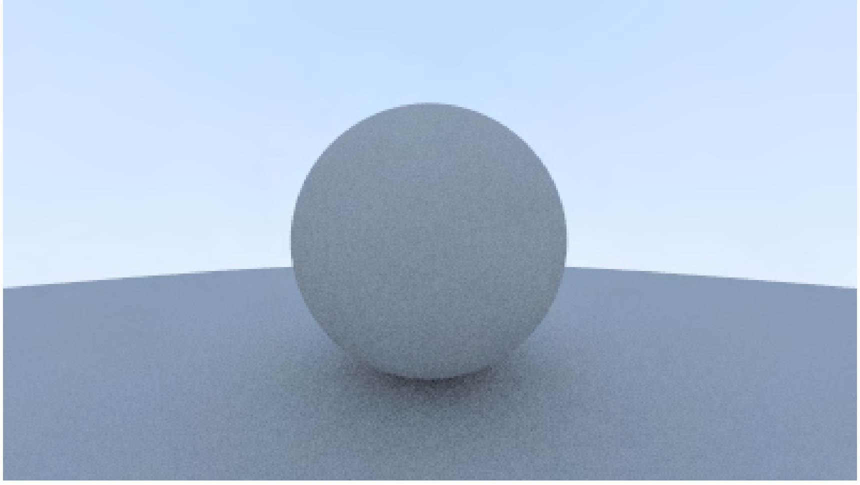


Figure 3.3: Correct rendering of Lambertian spheres

3.1.8 Metal

In the ninth chapter, titled ”Metal”, Peter Shirley introduces metallic materials and demonstrates how to implement them in the ray tracer. Metallic materials are important for creating realistic images, as they define how objects with reflective properties interact with light.

The author explains the physics of reflection and how to calculate the reflected direction of a ray. Unlike diffuse materials, which scatter light uniformly in all directions, metallic materials reflect light in a specific direction based on the incident ray’s angle and the surface normal, as shown in Figure 3.4.

Shirley demonstrates how to implement a metallic material by modifying the ”hittable” objects and the rendering loop. When a ray intersects an object with a metallic material, the ray tracer generates a new reflected ray, which is then traced recursively to calculate the final color contribution from the object. The author also introduces a fuzziness parameter to control the roughness of the metallic surface, which affects the sharpness of the reflection.

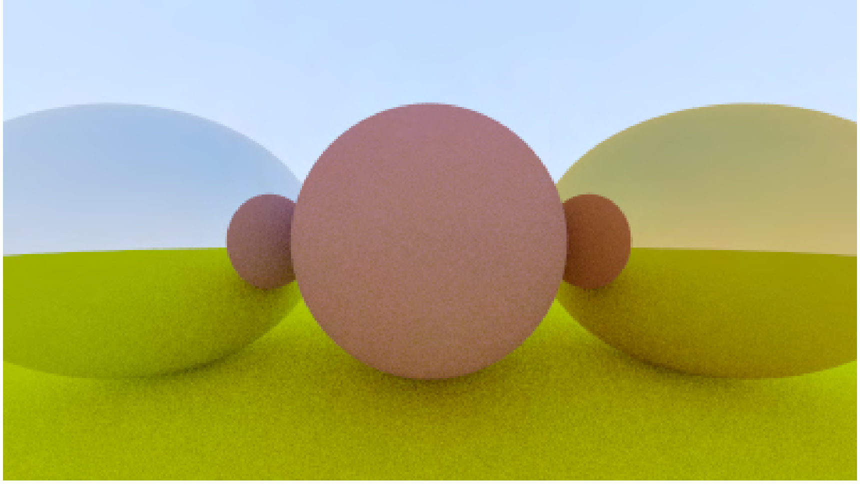


Figure 3.4: Shiny metal

3.1.9 Dielectrics

In the tenth chapter, titled ”Dielectrics”, Peter Shirley introduces dielectric materials, which are materials that transmit and refract light, such as glass or water. The author demonstrates how to implement dielectric materials in the ray tracer, which is essential for creating realistic images with transparent or refractive objects.

The chapter explains the physics of refraction and the Fresnel equations, which describe how light behaves when it passes through a dielectric material. The author also introduces Snell’s Law, a formula used to calculate the refracted direction of a ray when it passes from one medium to another with different indices of refraction.

Shirley demonstrates how to implement a dielectric material by modifying the ”hittable” objects and the rendering loop. When a ray intersects an object with a dielectric material, the ray tracer generates both refracted and reflected rays, depending on the angle of incidence and the Fresnel equations. The ray tracer then traces these rays recursively to calculate the final color contribution from the object.

A noteworthy observation regarding dielectric spheres is that employing a negative radius does not influence the geometry; however, it causes the surface normal to point inwards. This phenomenon can be utilized to create a hollow glass sphere, simulating the appearance of a bubble, as shown in Figure 3.5.

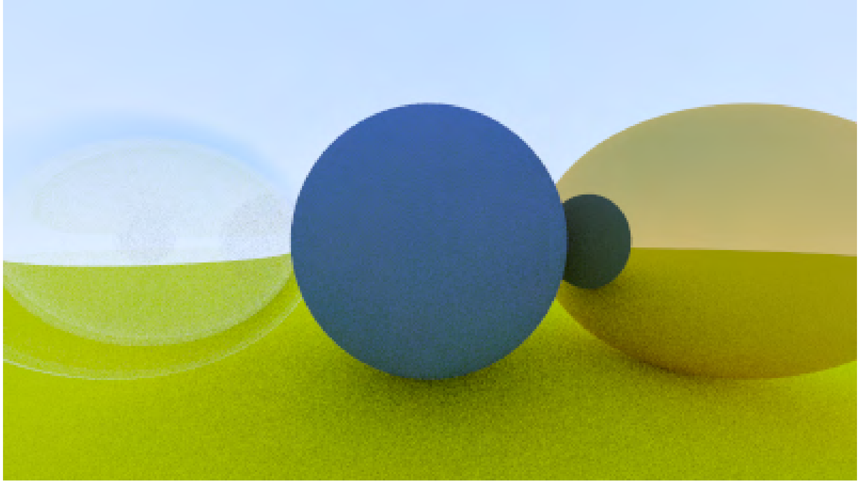


Figure 3.5: A hollow glass sphere

3.1.10 Positionable Camera

In the eleventh chapter, titled ”Positionable Camera”, Peter Shirley explains how to create a more flexible and positionable camera in the ray tracer. This enhancement allows for greater control over the viewpoint, orientation, and framing of the rendered scene, making it easier to achieve desired compositions and perspectives.

The author introduces several camera parameters, including position, target (look-at point), up vector, vertical field of view, and aspect ratio. By adjusting these parameters, the reader can control the camera’s location, direction, and field of view, which determine the final appearance of the rendered image.

Shirley demonstrates how to modify the existing camera implementation to account for these new parameters, enabling the camera to be easily positioned and oriented. The author also introduces the concept of the viewport and explains how to calculate its dimensions based on the field of view and aspect ratio.

3.1.11 Defocus Blur

In the twelfth chapter, titled ”Defocus Blur”, Peter Shirley introduces the concept of depth of field and demonstrates how to implement defocus blur in the ray tracer. Depth of field is an important aspect of photographic and cinematic imagery, where objects at different distances from the camera appear sharp or blurry depending on their relation to the camera’s focal plane.

The author explains that defocus blur is achieved by simulating a camera with a finite aperture, which causes rays to pass through different points on the aperture’s circumference. The degree of blur depends on the size of the aperture and the distance between the camera and the focal plane.

Shirley demonstrates how to modify the existing camera implementation to incorporate defocus blur. He introduces new camera parameters, such as aperture size and focus distance, and shows how to generate rays that pass through random points on the aperture. The modified camera model produces images with a more realistic depth of field effect, where objects in the scene exhibit varying degrees of sharpness and blur depending on their distance from the focal plane, as illustrated in Figure 3.6.

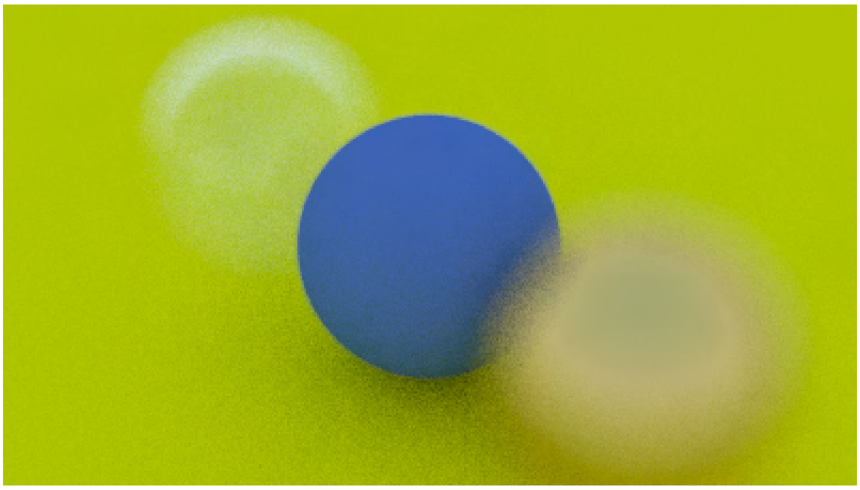


Figure 3.6: Spheres with depth-of-field

3.1.12 Where Next?

In the thirteenth chapter, titled ”Where Next?”, Peter Shirley concludes the book by discussing possible extensions, improvements, and further learning opportunities for readers who wish to continue their exploration of ray tracing.

Shirley highlights several areas in which the ray tracer can be enhanced, including:

- Performance optimization: The current implementation can be slow for complex scenes, so techniques such as bounding volume hierarchies (BVH), multithreading, and GPU acceleration can be explored to improve rendering performance.
- Advanced materials: Readers can experiment with more complex material models, such as subsurface scattering, anisotropic reflection, or physically based rendering (PBR) techniques.
- Texturing: Adding support for textures will allow objects in the scene to have more detailed and varied appearances, increasing the visual complexity of the rendered images.
- Lighting: Implementing more advanced lighting models, such as area lights or global illumination, can improve the realism of the ray tracer and the appearance of shadows and reflections in the scene.
- Volumes and Media: Adding support for volumetric effects, such as fog, smoke, or participating media, will enable more atmospheric and visually interesting scenes.

The final render of the book is illustrated in Figure 3.7.

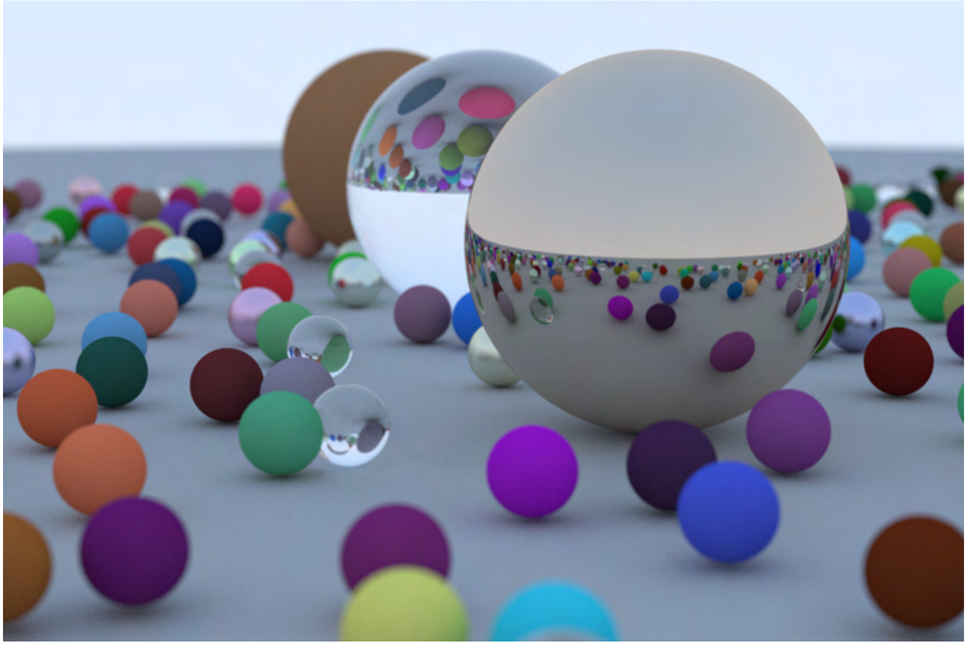


Figure 3.7: Final scene

3.2 Accelerated Ray Tracing In One Weekend in CUDA

In recent years, several projects have explored the potential of parallel computing and GPU acceleration for enhancing ray tracing performance. One of the most notable projects in this area is the ”Accelerated Ray Tracing in One Weekend in CUDA” blog post by NVIDIA [16], which demonstrates how to adapt the concepts from Peter Shirley’s ”Ray Tracing in One Weekend” book and implement them using the CUDA programming model.

The NVIDIA blog post offers a detailed explanation of the process of porting the original CPU-based ray tracer to a GPU-accelerated implementation using CUDA. The author starts by introducing the fundamentals of the CUDA programming model, including the concepts of threads, blocks, and grids, which are essential for understanding and exploiting GPU parallelism. These elements form the basis of the CUDA programming model, which allows developers to manage the execution of tasks on GPUs efficiently.

Next, the blog post delves into the implementation details of the GPU-accelerated ray tracer. The author outlines the necessary modifications and optimizations required to adapt the original ray tracer code for efficient execution on NVIDIA GPUs. These adaptations include:

- Data structures: Transitioning from standard C++ data structures to CUDA-compatible structures, such as using float3 instead of the custom vec3 class, for seamless GPU integration.

- Memory management: Implementing efficient memory management techniques for the GPU, including the use of global, constant, and shared memory spaces to optimize memory access patterns and reduce latency.

- Algorithm modifications: Refactoring the ray tracing algorithm to accommodate the constraints and advantages of the GPU architecture. This involves converting recursive functions into iterative loops, leveraging the GPU’s ability to handle large numbers of parallel threads.

- Load balancing: Ensuring that the computational workload is evenly distributed across the GPU’s streaming multiprocessors to maximize performance and avoid potential bottlenecks.

The blog post also addresses some of the challenges that arise when implementing a ray tracer on a GPU, such as the handling of recursion and the limited amount of available GPU memory. The author presents strategies for overcoming these challenges, including the use of an iterative approach for tracing rays and managing memory consumption by limiting the maximum depth of ray-object interactions.

In addition to the core implementation, the NVIDIA blog post highlights the significant performance gains achieved by using CUDA for ray tracing acceleration. The results showcase considerable reductions in rendering times compared to CPU-based implementations, making real-time rendering of complex 3D scenes a possibility.

Chapter 4

Real-Time Path Tracer Implementation

4.1 Introduction

In this chapter, we will delve into the implementation of our real-time path tracing application, which employs CUDA to accelerate ray tracing computations in 3D graphics. The foundation of our path tracer is based on the ”Ray Tracing in One Weekend” book series by Peter Shirley [15], which was discussed in greater detail in the previous chapter. The series provides a comprehensive and practical guide to implementing a basic ray tracer, which we have extended and optimized to harness the power of CUDA for real-time rendering.

The primary focus of this chapter is to present the process of constructing our real-time brute-force path tracer, detailing the various components and techniques employed. We will begin by discussing the various tools, technologies and libraries we used for setting up our project. Next, we will provide an overview of the path tracing algorithm we used in our implementation. Subsequently, we will outline the integration of CUDA with the path tracing algorithm, detailing the necessary modifications and optimizations required to achieve real-time performance. Furthermore, we will be discussing the capabilities of our real-time application, and how the user can interact with our engine during run-time, providing a detailed explanation of the user interface (UI) we built for that purpose. Last but not least, we will present an array of performance benchmarks pertaining to our application, demonstrating its efficacy and efficiency within the specified context.

Subsequently, we will outline the integration of CUDA with the path tracing algorithm, detailing the necessary modifications and optimizations required to achieve real-time performance. We will also discuss the implementation of various acceleration structures and techniques, such as bounding volume hierarchies (BVH) which are essential for improving the efficiency of our path tracer.

Throughout the chapter, we will demonstrate the practical aspects of our implementation, including code snippets and performance metrics, to provide a clear understanding of the techniques employed and their impact on rendering performance. By the end of this chapter, the reader will have gained valuable insight into the challenges and solutions involved in building a real-time path tracing application, with a particular focus on leveraging the power of CUDA for accelerated 3D graphics rendering.

4.2 Tools, Technologies and Libraries

In this section, we will provide an overview of the various tools, technologies, and libraries employed to set up and implement our real-time CUDA-accelerated path tracing project. These components have been carefully chosen to ensure seamless integration, efficient development, and optimal performance in our application.

4.2.1 Hardware Configuration

The development and performance testing of our ray tracing application were carried out on two different high-performance workstations to ensure broad hardware compatibility and robust performance.

System 1, which served as our primary development and benchmarking platform, was powered by an AMD Ryzen 5 3600 6-Core Processor operating at 3600 MHz. Accompanying the processor was 16 GB of DDR4 RAM, operating at 2133 MHz. For the GPU, we utilized an NVIDIA GeForce RTX 2060 Super, equipped with 8 GB of GDDR6 VRAM and 2176 CUDA cores, ideal for executing the massively parallel computations needed in our ray tracing application. This system was the primary platform on which we collected and analyzed our performance metrics for the thesis.

In parallel, we also used System 2 for development, to ensure our application's compatibility and performance on diverse hardware. This system featured an 11th Gen Intel Core i5-11600K processor, with 6 cores and a clock speed of 3912 MHz, paired with 16 GB of DDR4 RAM running at 3200 MHz. The GPU was an NVIDIA GeForce RTX 3070, boasting 8 GB of GDDR6 VRAM and a whopping 5888 CUDA cores, providing us with a highly capable platform to test the potential of our ray tracing application.

While the metrics reported in this thesis are primarily from System 1, the use of both systems during the development process ensured a broad perspective on performance and optimization strategies.

4.2.2 Development Environment and Language

Our project has been developed using the C++ programming language, which offers excellent performance, a rich ecosystem of libraries, and extensive support for parallel and concurrent programming. To ensure cross-platform compatibility, we have tailored our development environment for both Windows and Linux operating systems.

On Windows, we have relied on the Visual Studio Build Tools [17] for managing the build process, while using Visual Studio Code [18] as our primary code editor. Visual Studio Build Tools provide a streamlined build system for C++ projects, whereas Visual Studio Code offers a lightweight and versatile editing experience, complete with extensive support for plugins and language features.

For the Linux platform, we have utilized Neovim [19], a highly extensible and customizable text editor, enabling efficient development while maintaining platform-specific adaptability.

4.2.3 Debugger and Profiler Tools

In addition to the development environments and editors discussed previously, we have utilized a suite of debugger tools to facilitate efficient debugging, performance analysis, and optimization of our real-time CUDA-accelerated path tracing project. These tools have been instrumental in identifying bottlenecks, ensuring code correctness, and enhancing the overall performance of our application.

For debugging and profiling our GPU code, we have relied on NVIDIA's Nsight suite, which includes Nsight Compute [20] and Nsight Graphics [21]. Nsight Compute is a powerful GPU kernel profiler, allowing us to analyze and optimize the performance of our CUDA kernels, while Nsight Graphics provides invaluable insights into the graphics pipeline, enabling us to identify and resolve graphics-related issues.

To debug our application on the CPU side, we have used gdb, a widely adopted and versatile debugger for C++ applications. For debugging CUDA code on Linux, we have employed cuda-gdb, a GPU debugger specifically designed for CUDA applications, which extends the functionality of gdb for GPU debugging.

To further enhance our understanding of the application's performance, we have utilized Compute Sanitizer, a functional correctness checking suite for CUDA applications, which helps identify issues such as data races, memory leaks, and out-of-bounds memory access.

For profiling and optimizing our CPU code, we have leveraged AMD uProf [22] and Intel VTune [23]. AMD uProf is a performance analysis tool designed for AMD processors, providing valuable insights into CPU and memory usage, while Intel VTune is a performance profiler that offers a comprehensive analysis of CPU and system performance for Intel processors.

By incorporating these debugger and profiler tools into our development workflow, we have been able to effectively identify and resolve performance bottlenecks, ensure code correctness, and optimize our real-time CUDA-accelerated path tracing application to achieve maximum performance across various hardware configurations.

4.2.4 Glad for Generating OpenGL Functions

Glad is a powerful and flexible OpenGL function loader that generates function pointers for OpenGL functions at runtime [24]. We have chosen Glad for our real-time CUDA-accelerated path tracing project to ensure compatibility and maintainability across different OpenGL versions and platforms, allowing us to leverage the full potential of OpenGL in a streamlined manner.

One of the key advantages of using Glad is its ability to provide access to the latest OpenGL extensions and features, enabling our application to take advantage of cutting-edge graphics functionality. Glad generates a custom loader tailored to the specific OpenGL version and extensions requested during the configuration process, ensuring that our application remains up-to-date with the evolving OpenGL landscape.

Another important benefit of Glad is its platform-agnostic design, which simplifies the process of managing OpenGL function pointers across various operating systems and hardware configurations. This allows us to maintain a consistent development experience and runtime behavior on both Windows and Linux platforms without the need for platform-specific code.

Furthermore, Glad's lightweight and efficient implementation minimizes the overhead associated with OpenGL function loading, ensuring that our real-time path tracing application remains performant and responsive even when using advanced OpenGL features.

4.2.5 GLFW for Creating Windows, Contexts and Surfaces

GLFW is a widely-used library for creating windows, contexts, and surfaces in graphics applications [25]. In our real-time CUDA-accelerated path tracing project, we have utilized GLFW to handle the creation and management of our application's window and OpenGL context, ensuring a consistent and platform-independent foundation for our rendering pipeline.

A foremost advantage of using GLFW is its simple and portable API, which abstracts away the complexities of managing platform-specific windowing systems, allowing developers to focus on the core functionality of their applications. This has enabled us to maintain a clean and efficient development process, minimizing the need for platform-specific code and ensuring compatibility across various operating systems, including Windows and Linux.

In addition to window and context creation, GLFW provides an abstraction layer for handling input events, such as keyboard and mouse input. This has been invaluable in our project, as it allows us to implement user interactions and camera controls with ease, while maintaining a consistent input handling mechanism across different platforms.

GLFW also offers functionality for managing surfaces, which are essential for rendering our path-traced images to the screen. By leveraging GLFW's surface management capabilities, we have been able to efficiently display our rendered output, ensuring a smooth and visually-appealing user experience.

4.2.6 ImGui for Graphical User Interfaces (GUIs)

ImGui is a lightweight C++ library for creating immediate-mode GUIs [3], known for its simplicity, flexibility, and ease of integration into various projects. In our real-time CUDA-accelerated path tracing project, we have utilized ImGui to develop intuitive and user-friendly interfaces, enhancing the overall user experience and streamlining interaction with the application.

A prominent attribute of ImGui is its immediate-mode approach, which allows for efficient and dynamic UI updates without the need for complex state management. This approach enables our application to easily accommodate changes in rendering settings or scene configurations, providing real-time feedback and allowing users to fine-tune various aspects of the rendering process.

In addition to its core functionality, ImGui offers a wide range of built-in UI elements, such as sliders, checkboxes, and color pickers, which we have employed to create a comprehensive and versatile interface for our path tracing application. This has facilitated the customization of rendering parameters, camera controls, and scene objects, empowering users to explore different configurations and visualize their impact on the rendered output.

By incorporating ImGui into our project, we have been able to provide a rich and interactive UI for configuring rendering settings, managing scene objects, and navigating the application, all while maintaining optimal performance and minimizing the impact on our real-time path tracing implementation.

4.2.7 GLM for Vector Math

GLM is a C++ mathematics library designed for graphics software [26], providing a wide range of mathematical constructs and functions that are commonly used in graphics applications. We have used GLM to handle various mathematical operations, such as vector and matrix manipulation, throughout our path tracing project.

4.2.8 CUDA's Thrust Library

Thrust [27], a high-level CUDA library, has been incorporated for certain functions in our code. It offers a flexible interface for GPU programming, helping us manage memory and perform operations like sorting and searching, thereby making our code more concise and maintainable.

4.2.9 stb for Image Loading

The stb library [28] is a versatile collection of single-file, public domain libraries for C/C++ that offer a straightforward API for various tasks, including image file handling. In our real-time CUDA-accelerated path tracing project, stb has been chosen for handling image loading and saving operations due to its simplicity and ease of use.

An essential advantage of using stb is its minimalistic design, which allows for seamless integration into our project without introducing complex dependencies or bloating the codebase. This lightweight approach has enabled us to efficiently manage image-related operations while maintaining the performance and responsiveness of our application.

Stb supports a wide range of image formats, such as JPEG, PNG, and BMP, ensuring compatibility with a variety of input files and facilitating the import and export of rendered images. By leveraging stb's extensive format support, we have been able to accommodate diverse use cases and provide a robust image handling solution for our path tracing application.

4.2.10 spdlog for Logging

spdlog [29] is a high-performance, header-only C++ logging library that enables easy integration of logging functionality into applications, without introducing complex dependencies or incurring significant overhead. In our real-time CUDA-accelerated path tracing project, we have employed spdlog to facilitate efficient logging, debugging, and performance analysis across various stages of our development process.

One of the main advantages of using spdlog is its exceptional performance, which stems from its asynchronous and lock-free design. This allows for high-frequency logging with minimal impact on the overall performance of our path tracing application, ensuring that our logging activities do not interfere with real-time rendering and user interactions.

Another key benefit of spdlog is its extensive feature set, which includes support for various logging levels, customizable output formats, and integration with external log sinks, such as files or remote servers. By leveraging these features, we have been able to implement a flexible and comprehensive logging system that can be easily tailored to our specific requirements and adapted to different development and testing scenarios.

4.2.11 CMake and Premake for Building the Project

CMake [30] and Premake [31] are cross-platform build systems that generate platform-specific build files from simple configuration files. We have used both CMake and Premake to manage our project's build process, allowing for a streamlined and platform-agnostic build system.

By combining these tools, technologies, and libraries, we have established a robust development environment for our real-time CUDA-accelerated path tracing project, enabling efficient implementation and optimization of our path tracing algorithm while ensuring compatibility with a wide range of hardware and software configurations.

4.2.12 Project Folder Structure

A well-organized folder structure is crucial for maintaining the readability, modularity, and maintainability of any software project. In our real-time path tracing project, we have adopted a clear and logical folder structure that separates the various components of our application, facilitating efficient development, debugging, and collaboration.

Below is an overview of our project's folder structure:

```
CudaRayTracer/
    ├── assets/
    │     ├── fonts/
    │     ├── shaders/
    │     └── textures/
    ├── CudaRayTracer/
    │     ├── src/
    │     │     ├── Core/
    │     │     ├── Cuda/
    │     │     ├── Hittables/
    │     │     ├── ImGui/
    │     │     ├── Renderer/
    │     │     ├── Utils/
    │     │     └── Main.cpp
    │     ├── CMakeLists.txt
    │     └── premake5.lua
    ├── premake5-cuda/
    ├── scripts/
    │     ├── linux/
    │     └── windows/
    ├── vendor/
    │     ├── Glad/
    │     ├── GLFW/
    │     ├── glm/
    │     ├── ImGui/
    │     ├── premake/
    │     ├── spdlog/
    │     └── stb/
    ├── CMakeLists.txt
    ├── CudaRayTracerExternal.lua
    └── premake5.lua
```

This folder structure has been designed to promote modularity and separation of concerns, ensuring that each component of our application can be developed and tested independently, while also facilitating easy navigation and understanding of the project for both new and existing team members.

By adhering to a well-defined folder structure, we have been able to streamline our development process, improve the overall maintainability of our real-time CUDA-accelerated path tracing application, and create a solid foundation for future enhancements and optimizations.

4.3 Window Creation and Management with GLFW and OpenGL

One of the fundamental aspects of our project is the creation and management of the application window, for which we have leveraged the functionalities provided by the GLFW library and OpenGL.

4.3.1 Creating the Window

GLFW provides a robust and platform-independent API for creating windows, contexts, and surfaces, and for handling input and events. We utilized GLFW to create an OpenGL context and a window where our rendered images are displayed.

To achieve this, we first initialize GLFW and set the required window hints. These hints dictate certain properties about the window and the context, such as the OpenGL version and core profile.

Listing 4.1 is a snippet of code that initializes GLFW and sets the window hints.

```
if (!glfwInit())
{
    std::cerr << "Failed to initialize GLFW" << std::endl;
    return;
}

glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 4);
glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 5);
glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);
```

Listing 4.1: Initializing GLFW and setting window hints

Afterward, we create the window and the OpenGL context as seen in Listing 4.2.

```
GLFWwindow* window = glfwCreateWindow(800, 600, "CudaRayTracer", NULL,
    NULL);
if (!window)
{
    std::cerr << "Failed to create GLFW window" << std::endl;
    glfwTerminate();
    return;
}
```

```
glfwMakeContextCurrent(window);
```

Listing 4.2: Creating the window and the OpenGL context

4.3.2 Managing Inputs

Another critical aspect of any interactive application is input management. GLFW provides a simple way to handle user input, including mouse movements and keyboard inputs. In our path tracer, we have created callback functions to handle keyboard inputs, mouse movements, and window resize events. These inputs control various parameters, such as the camera's position and orientation, object transformation, and rendering options.

A small example of input management can be seen in Listing 4.3.

```
// Callback function for keyboard input
void key_callback(GLFWwindow* window, int key, int scancode, int action,
    int mods)
{
    if (key == GLFW_KEY_ESCAPE && action == GLFW_PRESS)
        glfwSetWindowShouldClose(window, GLFW_TRUE);
    // Add more keyboard controls here
}

// Callback function for mouse movement
void cursor_position_callback(GLFWwindow* window, double xpos, double
    ypos)
{
    // Handle mouse movement here
}

// Set the callbacks
glfwSetKeyCallback(window, key_callback);
glfwSetCursorPosCallback(window, cursor_position_callback);
```

Listing 4.3: Handling keyboard and mouse inputs

4.3.3 Clearing the Screen

Before each frame's rendering process, we clear the color buffer to prepare for the new frame. This is achieved with OpenGL's ***glClear*** function and ***GL_COLOR_BUFFER_BIT*** flag. It is essential to ensure that the buffer is clean before starting the rendering of a new frame to avoid visual artifacts.

In Figure 4.1 we can see the output of ***glClear*** in Listing 4.4.

```
glClearColor(0.4f, 0.4f, 0.4f, 1.0f);
glClear(GL_COLOR_BUFFER_BIT);
```

Listing 4.4: Clearing the color buffer

Figure 4.1: Window Creation and Clearing Color

After this, we are ready to start the rendering for the next frame, beginning with generating rays for each pixel.

These steps form the core loop of our application, initializing the window, processing inputs, clearing the color buffer, and then proceeding to the rendering operations. This loop continues until the application is closed by the user.

In the following sections, we'll discuss more advanced aspects of our application, including how we utilized CUDA and OpenGL to achieve real-time path tracing.

4.4 CUDA to OpenGL Interoperability

One of the key technical challenges we have addressed in our path tracing application is the efficient intercommunication between CUDA and OpenGL. This is vital to ensure that we can harness the computational power of CUDA for the computationally demanding task of path tracing, while simultaneously utilizing OpenGL for rendering the resulting image in real-time.

Traditional graphics processing would require data to be transferred from the GPU to the CPU and back to the GPU for rendering, which results in significant overhead due to the latency of these operations. To overcome this bottleneck, we exploit the interoperability features offered by CUDA [14], allowing CUDA kernels to directly write to OpenGL textures.

In our implementation, we have incorporated a technique inspired by NVIDIA's simpleCUDA2GL sample [13]. However, in contrast to the sample's use of Vertex Buffer Objects (VBOs), we have opted to register an OpenGL texture with CUDA. This alternative method provides the benefit of granting direct access to the texture memory, enabling CUDA to generate the image and OpenGL to directly render it, all while keeping the data on the GPU.

4.4.1 NVIDIA's simpleCUDA2GL Sample Approach

The traditional approach for CUDA-OpenGL interoperability, exemplified by NVIDIA's simpleCUDA2GL sample, is centered around OpenGL's Vertex Buffer Objects (VBOs). The sample showcases an approach where CUDA kernels write into OpenGL's Vertex Buffer Objects (VBOs). The steps involved in this procedure are as follows:

1. In Listing 4.5 an OpenGL buffer (VBO) is created with a specific size, usually proportional to the total number of pixels in the rendering context.

```
GLuint vbo;
glGenBuffers(1, &vbo);
glBindBuffer(GL_ARRAY_BUFFER, vbo);
glBufferData(GL_ARRAY_BUFFER, width * height * sizeof(float), NULL,
    GL_DYNAMIC_DRAW);
```

Listing 4.5: VBO Generation

2. The VBO is then registered with CUDA, marking it as a resource that can be accessed by CUDA kernels, as shown in Listing 4.6.

```
cudaGraphicsResource_t resource;
cudaGraphicsGLRegisterBuffer(&resource, vbo,
    cudaGraphicsRegisterFlagsNone);
```

Listing 4.6: VBO registration with CUDA

3. In Listing 4.7, before the CUDA kernel can use this resource, it has to be mapped, which returns a device pointer.

```
void *d_vbo_data;
cudaGraphicsMapResources(1, &resource);
cudaGraphicsResourceGetMappedPointer(&d_vbo_data, NULL, resource);
```

Listing 4.7: Mapping VBO to CUDA Kernel

This pointer can then be passed to the CUDA kernel, which fills the VBO with data.

4. After the CUDA kernel finishes its execution, the mapped resource must be unmapped, allowing OpenGL to regain exclusive access to the VBO, as shown in Listing 4.8.

```
cudaGraphicsUnmapResources(1, &resource);
```

Listing 4.8: Unmap Resources

5. The final step is for OpenGL to use the data that the CUDA kernel has written into the VBO for rendering. For instance, you may bind the VBO, set up the necessary attribute pointers, and issue a draw command like the Listing 4.9.

```
glBindBuffer(GL_ARRAY_BUFFER, vbo);
// Set up attribute pointers...
glDrawArrays(GL_TRIANGLES, 0, numberOfVertices);
```

Listing 4.9: OpenGL renders the VBO

4.4.2 Our Approach

While this procedure is efficient, it may not always be the optimal solution. In our application, we have implemented a different approach based on using an OpenGL texture rather than a VBO. This decision was driven by several reasons, the most significant of which is the inherent benefit of using texture memory: the ability to read data with hardware interpolation [32].

In the context of graphics programming, texture memory refers to a specific type of memory layout used by GPUs to store textures. Textures are essentially 2D (sometimes 3D or more) arrays of data that can be sampled (read from) in shaders. These shaders are small programs that run on the GPU and dictate how our scene gets rendered. Texture memory is optimized for this kind of sampling operation.

One critical aspect of texture memory is that it can use hardware interpolation. This feature means that when we read from a texture using non-integer coordinates, the GPU can automatically interpolate between the values of the surrounding cells. For example, if we try to sample a texture at the texture coordinate (0.5, 0.5), the GPU can automatically give us a value that is an average of the values at texture coordinates (0, 0), (1, 0), (0, 1), and (1, 1). This capability is often used to smoothly transition between different texture values when our object's surface aligns with the texture in a non-pixel-perfect way.

Now, coming to the context of our application, instead of using VBOs for storing data computed by CUDA, we are opting to use OpenGL textures. The implication is that instead of filling a VBO with data and using it directly for rendering, our CUDA kernel writes data into a texture. When it's time to render, instead of using a VBO to supply vertex data to a vertex shader, we use this texture as input to a fragment shader. This shader reads from the texture (sampling it), and uses the sampled values to determine the color of each pixel on the screen.

Here is a breakdown of our approach:

1. In Listing 4.10, instead of a VBO, we create an OpenGL texture with the same dimensions as our rendering context.

```
GLuint textureID;
glGenTextures(1, &textureID);
glBindTexture(GL_TEXTURE_2D, textureID);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
     // clamp s coordinate
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
     // clamp t coordinate
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA8, width, height, 0, GL_RGBA,
    GL_UNSIGNED_BYTE, NULL);
```

Listing 4.10: OpenGL Texture Creation

2. Similar to the previous procedure, Listing 4.11 shows how the texture is then registered with CUDA.

```
cudaGraphicsResource_t resource;
```

```
cudaGraphicsGLRegisterImage(&resource, textureID, GL_TEXTURE_2D,
    cudaGraphicsRegisterFlagsWriteDiscard);
```

Listing 4.11: Texture CUDA Registration

3. The mapping step is slightly different in this case. Instead of obtaining a device pointer, we obtain a ***cudaArray*** pointer, which then is bound to a texture reference in the CUDA kernel. An example of this procedure can be seen in Listing 4.12.

```
cudaArray_t textureArray;
cudaGraphicsMapResources(1, &resource);
cudaGraphicsSubResourceGetMappedArray(&textureArray, resource, 0, 0)
    ;
```

Listing 4.12: CUDA Texture Mapping

4. Following the execution of the CUDA kernel and the data write into the texture memory, the resource is unmapped in Listing 4.13.

```
cudaGraphicsUnmapResources(1, &resource);
```

Listing 4.13: CUDA Texture Unmapping

5. The final step is to bind the texture and draw a fullscreen quad, effectively displaying the image that was written into the texture by our CUDA kernel. Below is an overview of the steps you would typically take to render a full-screen quad with a texture applied to it:

   (α′) Set up your vertex and fragment shaders. The vertex shader would usually be very simple, only responsible for transforming your quad vertices to cover the full screen. The fragment shader, on the other hand, would sample from the bound texture and return the sampled color for each pixel. An example of a GLSL [33] vertex and fragment shader is shown in Listing 4.14

```
// Vertex Shader
#version 330 core
in vec2 aPos;
in vec2 aTexCoords;

out vec2 TexCoords;

```

```
void main()
{
    gl_Position = vec4(aPos.x, aPos.y, 0.0, 1.0);
    TexCoords = aTexCoords;
}

// Fragment Shader
#version 330 core
in vec2 TexCoords;
out vec4 FragColor;

uniform sampler2D screenTexture;

void main()
{
    FragColor = texture(screenTexture, TexCoords);
}
```

Listing 4.14: Vertex and Fragment Shader

(β′) Create and bind a Vertex Array Object (VAO) for your full-screen quad. This VAO would typically contain two VBOs, one for the quad's vertices and one for their associated texture coordinates, like in Listing 4.15.

```
float quadVertices[] = {
    // positions   // texCoords
   -1.0f,  1.0f,  0.0f, 1.0f,
   -1.0f, -1.0f,  0.0f, 0.0f,
    1.0f, -1.0f,  1.0f, 0.0f,

   -1.0f,  1.0f,  0.0f, 1.0f,
    1.0f, -1.0f,  1.0f, 0.0f,
    1.0f,  1.0f,  1.0f, 1.0f
};
```

Listing 4.15: Quad Vertices

(γ′) Bind your texture and render the full-screen quad. At this point, you would bind the texture object that contains the data written by your CUDA kernel, activate your shaders, and finally draw your full-screen quad. The end result would be that your texture gets displayed on the full screen, as shown in Listing 4.16.

```
glBindTexture(GL_TEXTURE_2D, textureID);
// assuming you have the shader program id and the VAO id
glUseProgram(shaderProgram);
glBindVertexArray(VAO);
glDrawArrays(GL_TRIANGLES, 0, 6);
```

Listing 4.16: Render The Fullscreen Quad

In Figure 4.2 we can observe the above implementation.

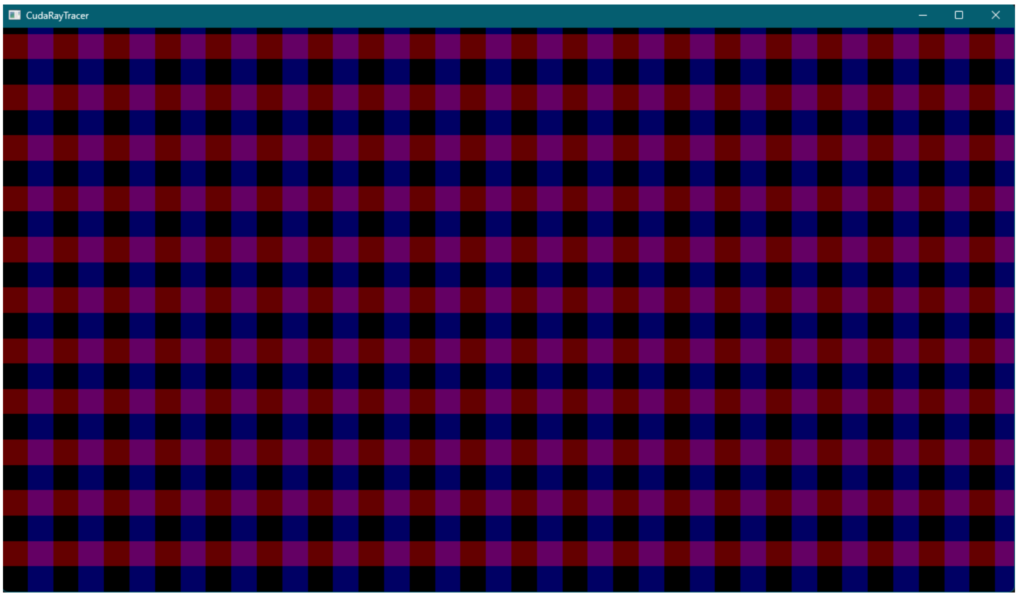


Figure 4.2: CUDA Texture Rendered on a 2D Quad

This alternative technique allows us to not only minimize data transfer, but also enables hardware interpolation of texture memory when read from the CUDA kernels, thus providing performance optimization. This method also enabled us to render the output directly into an ImGui Image window, instead of the fullscreen quad we mentioned before. More on that later...

By implementing this modified CUDA-OpenGL interoperability technique, we've successfully optimized our path tracer to generate and display images in real-time. In subsequent sections, we will delve into further intricacies of our project, including the implementation of ray-tracing algorithms, mathematical operations required, and further use of CUDA for acceleration.

4.5 Integrating ImGui for User Interface and Texture Display

To ensure usability and provide a user-friendly interface for our application, we integrated the Dear ImGui library [3], often referred to as ImGui. ImGui is an immediate-mode graphical user interface library that is lightweight, code-based, and predominantly employed in content creation tools and debug tools in the game development industry due to its simplicity and ease of integration.

ImGui shines when it comes to rendering hardware-accelerated textures, which is critical for our application. After registering the OpenGL texture with CUDA and performing the necessary computations (as detailed in the previous sections), we needed to display the result inside an ImGui window for users to see the results of the ray tracing in real time.

The procedure to render the CUDA-processed OpenGL texture inside an ImGui window is straightforward [34]. After creating a new ImGui window, we used the ***ImGui::Image*** function, which accepts an ***ImTextureID*** and the size of the image to be rendered, as shown in Listing 4.17.

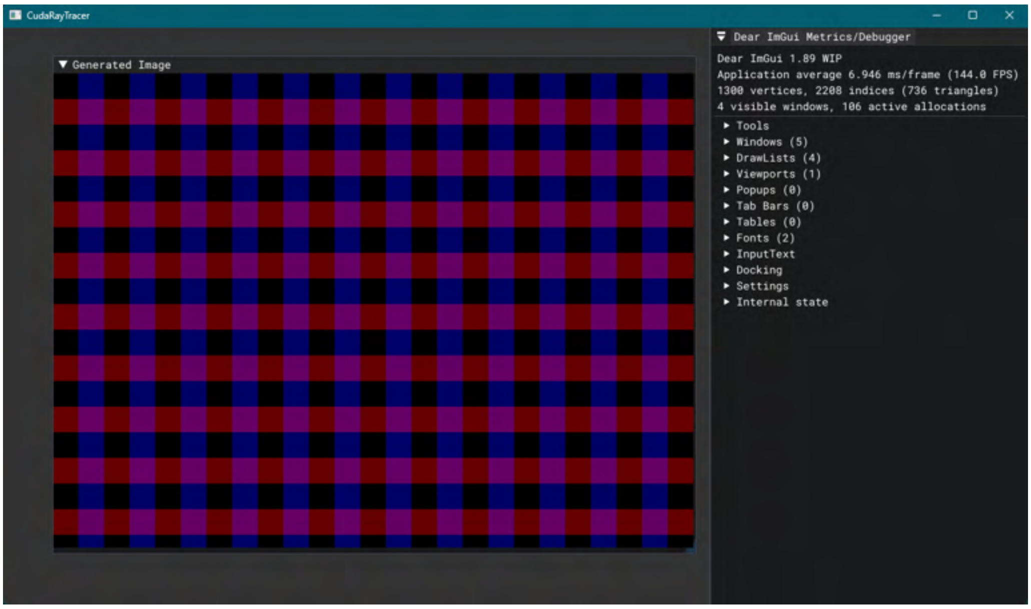
```
// Suppose 'textureID' is the ID of the texture you want to display
// and 'tex_width' and 'tex_height' are its dimensions
ImGui::Begin("Generated Image");
ImGui::Image((void*)(intptr_t)textureID, ImVec2(tex_width, tex_height),
    ImVec2(0, 1), ImVec2(1, 0));
ImGui::End();
```

Listing 4.17: ImGui Image Loading

Here, ***ImTextureID*** is a typedef for ***void****, so we just need to cast our OpenGL texture ID into this type. The two ***ImVec2*** parameters define the UV coordinates for the texture mapping. Please note that we have flipped the y-coordinates because OpenGL textures are usually y-flipped. An example of the CUDA generated image displayed in an ImGui window is shown in Figure 4.3.

This approach allows us to seamlessly integrate our CUDA-accelerated ray tracing with a UI layer. Users can observe the real-time rendering results, and we can extend this UI to include various user controls for adjusting rendering parameters, pausing and resuming rendering, and other features, making our application interactive and user-friendly.

In our application, the ImGui rendering functions were invoked within each rendering



```
Dear ImGui 1.89 WIP
Application average 6.946 ms/frame (144.0 FPS)
1300 vertices, 2208 indices (736 triangles)
4 visible windows, 106 active allocations
```

Figure 4.3: CUDA Generated Image displayed in an ImGui Window

loop iteration. This procedure ensured that the most up-to-date graphics context and framebuffer status were used when rendering the ImGui interface. This way, the ImGui overlay consistently reflects the latest state of our CUDA-accelerated ray tracing algorithm.

As ImGui is a backend-agnostic library, its setup requires specific bindings that match the underlying graphics API used. For our project, given our reliance on OpenGL and GLFW, we initialized ImGui with the appropriate OpenGL3 and GLFW backend bindings. This configuration enabled ImGui to operate effectively within our application's graphical environment.

It is worth emphasizing that this design choice ensures the consistency of our application's interface and enhances its robustness. By incorporating ImGui into our application, we effectively melded high-performance real-time ray tracing with an intuitive and responsive user interface. This combination elevated the user experience significantly by allowing for interactive engagement and on-demand visual feedback.

To summarize, ImGui not only provides a platform for displaying our real-time ray tracing results in a more accessible and intuitive format, but it also paves the way for potential enhancements and extensions to our application. This ImGui integration ensures our application is not just about performance and quality of results, but also about user experience and ease of use.

4.6 Ray Tracing Implementation

Our path tracing implementation is fundamentally inspired by and structured around the models and methodologies put forth in Peter Shirley's influential ”Ray Tracing in One Weekend” book series [15], as we meticulously described in section 3.1.

In our application, these principles serve as the foundational blueprint upon which we built and enhanced our ray tracer. Our focus was to develop a GPU-accelerated, real-time application capable of rendering complex 3D scenes with high fidelity. We adopted the basic structure provided in the series, and then we optimized it using CUDA and other modern programming techniques.

In the following sections, we delve into the core mathematical and computational aspects of our ray tracing implementation. This includes an exploration of vector operations, the concept of rays and how they're utilized, the handling of intersections between rays and 3D objects, the process of implementing a virtual camera, and the algorithms for simulating the behavior of light as it interacts with different surfaces, our implementation of the BVH algorithm and image texture mapping.

While our primary reference remains ”Ray Tracing in One Weekend”, our real-time ray tracer surpasses the original minimalist design proposed by Shirley. It demonstrates how one can start from a simple, bare-bones ray tracer and augment it to create an application capable of producing stunning, photorealistic renderings in real time.

4.7 Ray Generation and Virtual Camera

One of the most essential components in ray tracing is the ray itself. In terms of mathematical representation, a ray $R(t)$ in 3D space is described parametrically as:

$$R(t) = A + t * B \tag{4.1}$$

where A signifies the ray's origin, B represents the direction vector, and t is a scalar parameter that stretches or shrinks B.

The ***Ray*** class defines a ray in the context of our algorithm. The class comprises two primary elements: a 3D point for the ray's origin and a 3D vector for the ray's direction. This class also contains a method that computes a point along the ray for a given parameter t, reflecting our mathematical definition of a ray.

When our ray tracer creates an image, it generates rays from the camera's location, with each ray corresponding to a pixel in the final image. The process of ray generation is integral to the operation of the ray tracer.

To start, we set up an ***origin*** vector and three basis vectors representing the camera's orientation: ***forward***, ***up***, and ***right***. The origin is the position of the camera in the 3D world, while ***forwardV***, ***upV***, and ***rightV*** form a coordinate system relative to the camera's perspective.

In Listing 4.18, is an example of setting up the camera system and generating rays based on the camera's position.

```
Vec3 col = Vec3(0.0f, 0.0f, 0.0f);

Vec3 origin = Vec3(origin_x, origin_y, origin_z);
Vec3 forwardV = Vec3(orientation_x, orientation_y, orientation_z);
Vec3 upV = Vec3(up_x, up_y, up_z);
Vec3 rightV = Normalize(Cross(upV, forwardV));
Vec3 center = Vec3(width / 2.0f, height / 2.0f, 0.0f);

float u = (float)(x - center.x()) / (float)(width);
float v = (float)(center.y() - y) / (float)(width);
Vec3 distFromCenter = (u * rightV) + (v * upV);
Vec3 startPos = (near_plane * distFromCenter) + origin + (fov * forwardV);
Vec3 secondPlanePos = (far_plane * distFromCenter) + ((1.0f / fov * 10.0f) * forwardV) + origin;
Vec3 dirVector = Normalize(secondPlanePos - startPos);

Ray r = Ray(startPos, dirVector);
col = col + color(r, world, max_depth);
```

Listing 4.18: Ray Generation and Camera

In the above Listing, ***center*** is a point representing the center of the image plane. For each pixel in the image, we calculate its relative position on the image plane using uv coordinates. Here, u and v represent the horizontal and vertical offsets from the center of the image plane, respectively. The ***distFromCenter*** vector represents the displacement from the center of the image plane to the uv coordinate.

We then calculate the starting position ***startPos*** of the ray by moving from the camera's origin along the forward vector by the focal length (***fov***). This position is further offset by the

distFromCenter vector, scaled by the near plane distance (***near_plane***).

The ray's direction (***dirVector***) is computed as a normalized vector pointing from ***startPos*** to ***secondPlanePos***. ***secondPlanePos*** is a point located on a secondary image plane farther from the origin along the forward vector, offset by ***distFromCenter*** scaled by the far plane distance (***far_plane***).

Once the starting position and direction are determined, we can create a new Ray object. The ray tracer then colors each pixel by calling the color function, which sends the ray into the world and computes the ***color*** that the ray encounters.

The ***color*** function, determines the color that a ray would see when cast into the scene. The color returned by this function is dependent on the objects the ray interacts with in the scene. We will talk about the ray-object intersection in the next section.

If the ray does not hit an object, it calculates the color based on a linear blend of two other colors. This process is often referred to as linear interpolation or lerp for short. In this context, the lerp operation serves to simulate a simple gradient sky.

The above methodology encapsulates the core logic of ray generation in the context of a virtual camera within our ray tracer. Through this process, the ray tracer can simulate the intricate interplay of light and materials within a 3D scene to create highly realistic images.

4.8 Ray-Object Intersections

At the heart of any ray tracing algorithm is the concept of ray-object intersection. This process forms the basis of the recursive ray tracing algorithm, determining whether a ray intersects with an object in the scene, and if so, where that intersection point is.

Our application primarily focuses on intersecting rays with two types of geometric shapes: spheres and rectangles. Spheres, due to their mathematical elegance, provide an excellent introductory example, effectively encapsulating the fundamental principles of ray tracing. Rectangles, on the other hand, introduce a layer of complexity and provide the groundwork for extending the algorithm to arbitrary polygons.

A sphere in 3D space is described by an equation of the form:

$$(P - C) \cdot (P - C) = r^2$$

where P represents any point on the sphere, C is the center of the sphere, r is the radius, and $\cdot$ denotes the dot product.

To find the intersection of the ray and the sphere, we substitute the equation of the ray 4.1 into the sphere equation, resulting in a quadratic equation of the form:

$$(A + t * B - C) \cdot (A + t * B - C) - r^2 = 0$$

The roots of this equation give us the possible values of t where the ray intersects the sphere. If the roots are imaginary, the ray does not intersect the sphere. If there is one root, the ray is tangent to the sphere at one point. If there are two roots, the ray passes through the sphere, intersecting it at two points.

A key aspect of our implementation is dealing with these intersection points. For each intersection, we calculate the corresponding point on the sphere and the normal at that point. This information is used in the shading process, which relies on the incident angle of the ray and the normal at the intersection point.

For the rectangles, we used axis-aligned rectangles, which restrict the rectangle's orientation to align with the Cartesian coordinate axes. This simplification substantially reduces the computational complexity of checking whether the intersection point lies within the rectangle boundaries.

To implement an axis-aligned rectangle, we first define its position and extent along two axes, while the third axis position is kept constant. For instance, an x-y rectangle is defined by its x and y extents $[x0, x1]$ and $[y0, y1]$, and a constant z-coordinate $(z0)$. The rectangle equation for this case is:

$$x0 \leq x \leq x1, y0 \leq y \leq y1, z = z0$$

To find the intersection of a ray with this rectangle, we substitute the ray equation 4.1 into the rectangle's:

$$A + t * B = P$$

And solve for t:

$$t = \frac{(z0 - A.z)}{B.z}$$

The intersection point P is then obtained by substituting t back into the ray equation. To check if the intersection point lies within the rectangle's boundaries, we simply ensure the x and y coordinates of P are within the limits $[x0, x1]$ and $[y0, y1]$, respectively. If these conditions are met, the ray intersects the rectangle; otherwise, it does not.

Similarly, y-z and z-x rectangles can be defined and tested for intersection in the same manner, with the checks and constant coordinate adjusted accordingly.

This streamlined intersection approach is a significant advantage when dealing with axis-aligned rectangles, improving the efficiency of the ray tracing algorithm while maintaining accurate renderings. Despite the constraint on orientation, this method provides an excellent way to create a wide range of complex scenes.

The ray-object intersection is a critical part of our application. We employed an object-oriented approach without virtual functions to address the limitations of CUDA.

Our ray-object intersection algorithm is encapsulated within two main constructs: the ***Ray*** class 4.7 and the ***Hittable*** class.

The ***Hittable*** class acts as a general base class for all scene objects that a ray could potentially intersect. This class does not employ virtual functions, given CUDA's limitations with respect to polymorphism. Instead, a ***HittableType*** enum and a union, ***ObjectUnion***, are utilized to differentiate and handle various object types.

The ***HittableType*** enum lists the potential object types we deal with in our application - spheres, rectangles in XY, XZ, and YZ planes. The ***ObjectUnion*** union, on the other hand, holds a pointer to the specific object instance, allowing for efficient memory usage.

When an intersection check is required, the ***HittableType*** determines which type of object is being handled, and the corresponding member of the ***ObjectUnion*** is accessed for detailed intersection logic specific to that object type.

In essence, the ***Hittable*** class is instrumental in managing the ray-object intersection checks in a flexible and efficient manner, maintaining our code's clarity and extensibility while complying with the constraints of CUDA. This design has helped encapsulate the complexities of the ray-object intersection process and keep the rest of our program separated from these intricacies.

In conclusion, while our ray tracing application predominantly implements intersections with spheres and rectangles, the essential principle remains the same: to find if and where a ray intersects an object. As a result, it can be extended to include a wide variety of geometric shapes, each with its specific intersection algorithm.

4.9 Surface Normals

Surface normals play a pivotal role in computer graphics and, in particular, in ray tracing algorithms. They are essentially vectors that are perpendicular to an object's surface at the point of intersection. Understanding the direction in which these vectors point is crucial for determining how light should scatter or reflect when it hits an object, which in turn is critical for generating realistic images.

In the case of a sphere, calculating the surface normal at a given intersection point is quite straightforward due to the geometrical properties of the shape. If P represents a point on the sphere's surface and C is the sphere's center, the vector from C to P will always point directly outwards, forming a right angle with the tangent plane to the sphere at P. This makes it ideal as a normal vector. However, we typically want our normal vectors to be ”unit vectors”, i.e., vectors of length 1, for ease of computation later on. To achieve this, we can normalize the vector $(P - C)$ by dividing it by the radius of the sphere, r, resulting in the formula:

$$\frac{(P - C)}{r}$$

In the case of rectangles, the surface normal is constant and depends on the plane where the rectangle lies. For example, in an XY rectangle, the surface normal would be either $(0, 0, 1)$ or $(0, 0, -1)$, i.e., along the positive or negative Z-axis. The choice between the two depends on the orientation of the rectangle, which side should be considered as ”front”, a concept that becomes important when dealing with issues such as backface culling.

In our implementation, each object type is responsible for calculating the surface normal at its intersection point with a ray. The calculated normal is then returned along with other intersection details, ready to be used in subsequent computations, especially those concerning lighting and shading, which heavily rely on the surface normal.

Taking into account the direction of the incoming ray is crucial when returning the normal direction, as we always want the normal to point against the ray. This is to ensure the correct determination of the shaded and lit sides of the object when dealing with enclosed objects or those with an inside and an outside, like a sphere.

In conclusion, the accurate calculation and utilization of surface normals form the foundation of our shading computations, providing a means to accurately model how light interacts with the surfaces in our scene.

4.10 Materials

Materials in computer graphics define how surfaces interact with light, producing diverse visual effects that contribute significantly to the realism and aesthetic quality of rendered images. In our ray tracer, we've chosen to focus on implementing four key types of materials, each with distinct properties and behaviors when interacting with light. These are: Lambertian, Metal, Dielectric, and Diffuse Light materials. This selection enables us to model a wide range of common physical materials, ranging from diffuse surfaces like chalk or unglazed pottery (Lambertian), to reflective surfaces like polished metal (Metal), transparent materials like glass or water (Dielectric), and objects that emit light (Diffuse Light). By implementing these materials, we provide our ray tracer with the versatility to accurately represent and simulate an expansive array of real-world scenes.

4.10.1 Lambertian

Lambertian materials, so named after the physicist Johann Heinrich Lambert, are designed to emulate perfectly diffuse surfaces — surfaces that scatter light uniformly in all directions. Examples of such surfaces in the real world include materials like chalk or unglazed pottery. The characteristic property of these surfaces is that their brightness appears consistent regardless of the viewing angle. This uniform distribution of light, irrespective of the observer's viewpoint, is central to the Lambertian reflectance model [35].

The mathematical representation of Lambertian reflection is a cosine distribution over the hemisphere. Given that the cosine of an angle decreases as the angle increases, rays scattered near the normal direction are given higher weight. However, generating random points in cosine-weighted hemisphere might be computationally intensive. Therefore, we adopt an alternative approach where we scatter rays in random directions within a unit sphere, and if the scattered ray is in the opposite direction to the normal, we negate it. This ensures that the scattered rays always lie in the same hemisphere as the surface normal.

In the context of our ray tracer, when a ray intersects with an object possessing a Lambertian material, we generate a scattered ray in this random direction. The color of the scattered ray is obtained by scaling the incoming ray's color by the material's albedo (color). This scattered ray is then sent back into the ray tracer to compute the color contribution from reflections.

In Listing 4.19, we provide some pseudocode to illustrate the above concept. Keep in mind, this pseudocode is meant to give a high-level understanding of the process and does not correspond exactly to our actual implementation.

```
// Structure for Lambertian Material
struct Lambertian {
    Vec3 albedo;  // albedo color of the material
};

// Function to generate a random direction within a unit sphere
Vec3 random_in_unit_sphere() {
    Vec3 p;
    do {
        p = 2.0*Vec3(drand48(), drand48(), drand48()) - Vec3(1,1,1);
    } while (p.squared_length() >= 1.0);
    return p;
}

// Scattering function for Lambertian material
bool scatter(const Ray& r_in, const HitRecord& rec, Lambertian& mat, Vec3& attenuation, Ray& scattered) {
    Vec3 target = rec.p + rec.normal + random_in_unit_sphere();
    scattered = Ray(rec.p, target-rec.p);  // New scattered ray
    attenuation = mat.albedo;  // The color is scaled by the material’s albedo
    return true;
}
```

Listing 4.19: Lambertian Material

In the Listing above, ***drand48()*** generates a random double between 0.0 and 1.0. The ***scatter*** function takes as input the incoming ray ***(r_in)***, the hit record ***(rec)***, and the Lambertian material ***(mat)***, and computes the scattered ray and its attenuation. The ***HitRecord*** structure contains information about the intersection point, such as the point itself ***(rec.p)*** and the normal at that point ***(rec.normal)***. If the scattering is successful (which is always the case for Lambertian materials), it returns true.

For a Lambertian surface, the new scattered direction is computed as the sum of the hit point, the normal at the hit point, and a random point inside a unit sphere. The scattered ray is then created with the hit point as the origin and the new direction as its direction. The color of the scattered ray (the ***attenuation***) is the color of the incoming ray scaled by the material's albedo.

The use of Lambertian materials is an effective way to model the interaction between light and matte surfaces. Its mathematical and conceptual simplicity make it an excellent tool for approximating the diffuse reflection phenomenon that is pervasive in the real world, thereby contributing to the realism of the rendered scenes in our application.

4.10.2 Metal

Metal materials provide a different type of reflection, where light tends to be reflected in a single primary direction, rather than being scattered in all directions as with Lambertian materials. The reflectance characteristic of metal materials is typically based on the concept of specular reflection [36].

Specular reflection follows the law of reflection, which states that the angle of incidence is equal to the angle of reflection. Here, the angle of incidence is the angle between the incoming ray and the surface normal, while the angle of reflection is the angle between the reflected ray and the surface normal.

Moreover, real-world metals aren't perfectly smooth and might reflect rays in slightly different directions. This ”roughness” is simulated by adding a fuzziness factor to the reflected direction, scattering the ray in a small cone around the perfect reflection direction.

In Listing 4.20, is some pseudocode illustrating the above.

```
// Structure for Metal Material
struct Metal {
    Vec3 albedo;  // albedo color of the material
    float fuzz;   // fuzziness factor of the material
};

// Function to reflect a vector v about a normal n
Vec3 reflect(const Vec3& v, const Vec3& n) {
    return v - 2*dot(v,n)*n;
}

// Scattering function for Metal material
bool scatter(const Ray& r_in, const HitRecord& rec, Metal& mat, Vec3&
   attenuation, Ray& scattered) {
    Vec3 reflected = reflect(normalize(r_in.direction()), rec.normal);
    scattered = Ray(rec.p, reflected + mat.fuzz*random_in_unit_sphere());
    attenuation = mat.albedo;
    return (dot(scattered.direction(), rec.normal) > 0);
}
```

Listing 4.20: Metal Material

In the above Listing, ***scatter*** function first computes the perfect reflection direction using the ***reflect*** function and then adds a random direction within a sphere of radius equal to the material's fuzziness. The scattered ray is then created with the hit point as the origin and this new direction. The color of the scattered ray (the ***attenuation***) is the color of the incoming ray scaled by the material's albedo.

The function returns ***true*** only if the scattered direction is in the same hemisphere as the surface normal. This check ensures that rays aren't scattered below the surface.

4.10.3 Dielectrics

Dielectric materials, representing transparent objects like glass or water, present a more complex interaction with light rays, involving both reflection and refraction. Refraction is the bending of the light path when it passes from one medium into another. The amount of bending is determined by the indices of refraction of the two media and the angle of incidence.

The Fresnel equations [37], named after Augustin-Jean Fresnel, describe how much light is reflected and how much is refracted at the interface between two media. However, these equations can be quite complex, especially for an iterative process like ray tracing. Therefore, we use Schlick's approximation [38], a simplified model that approximates the Fresnel equations with minimal performance cost.

Additionally, it's important to account for total internal reflection (TIR). TIR happens when light traveling in a medium with a high index of refraction strikes the boundary with a medium of lower index of refraction at a sufficiently high angle; in such cases, all the light is reflected, and none is refracted.

In Listing 4.21, is an overview of the above process in pseudocode form.

```
// Structure for Dielectric Material
struct Dielectric {
    float ref_idx;  // refractive index of the material
};
```

```

// Function to refract a vector v into a medium with normal n and relative refractive index etai_over_etat
Vec3 refract(const Vec3& v, const Vec3& n, float etai_over_etat) {
    float cos_theta = dot(-v, n);
    Vec3 r_out_perp =  etai_over_etat * (v + cos_theta*n);
    Vec3 r_out_parallel = -sqrt(fabs(1.0 - r_out_perp.length_squared())) * n;
    return r_out_perp + r_out_parallel;
}

// Scattering function for Dielectric material
bool scatter(const Ray& r_in, const HitRecord& rec, Dielectric& mat, Vec3& attenuation, Ray& scattered) {
    attenuation = Vec3(1.0, 1.0, 1.0);
    float etai_over_etat = (rec.front_face) ? (1.0 / mat.ref_idx) : mat.ref_idx;

    Vec3 unit_direction = normalize(r_in.direction());
    float cos_theta = fmin(dot(-unit_direction, rec.normal), 1.0);
    float sin_theta = sqrt(1.0 - cos_theta*cos_theta);

    if (etai_over_etat * sin_theta > 1.0 ) {  // Handle total internal reflection
        Vec3 reflected = reflect(unit_direction, rec.normal);
        scattered = Ray(rec.p, reflected);
        return true;
    }

    Vec3 refracted = refract(unit_direction, rec.normal, etai_over_etat);
    scattered = Ray(rec.p, refracted);
    return true;
}
```

Listing 4.21: Dielectric Material

In the above Listing, we first calculate whether total internal reflection occurs. If it does, we treat the interaction as a reflection. If not, we compute the refracted direction using the ***refract*** function and treat the interaction as refraction. As before, the ***scatter*** function returns

true to indicate a scattering event, with the scattered ray and attenuation color updated for the subsequent path segment.

4.10.4 Diffuse Light

Diffuse Light materials simulate objects that act as light sources. Unlike the materials discussed above, these materials do not react to incoming rays. Instead, they constantly emit light, meaning they radiate a fixed color in all directions.

This kind of material is an important feature in rendering realistic scenes, as it enables the modeling of light sources such as lamps, monitors, and the sun, enhancing the authenticity of the scene illumination.

In our implementation, a Diffuse Light material is represented by a texture that provides the emitted color. When a ray hits an object with this material, instead of generating a scattered ray, we check whether the ray was hit from the front side (to avoid illuminating the back sides of the objects) and return the color of the texture at the hit point.

In Listing 4.22, is shown some pseudocode for the Diffuse Light material.

```
// Structure for Diffuse Light Material
struct DiffuseLight {
    Texture emit;  // emission texture
};

// Emission function for Diffuse Light material
Vec3 emitted(const DiffuseLight& mat, float u, float v, const Vec3& p) {
    return mat.emit.value(u, v, p);
}

// Scattering function for Diffuse Light material
bool scatter(const Ray& r_in, const HitRecord& rec, DiffuseLight& mat,
   Vec3& attenuation, Ray& scattered) {
    return false;  // Diffuse Light does not scatter rays
}
```

Listing 4.22: Diffuse Light Material

In the above Listing, we first check if the ray is hitting the surface from the front side (i.e., in the direction opposite to the surface normal). If it is, we return the value of the emission texture at the hit point. Otherwise, we return a zero vector, meaning no light is emitted. The

scatter function simply returns ***false*** to indicate that no scattering occurs.

As can be seen, this is a considerable departure from the previous material types, which all involved some form of scattering. With Diffuse Light, we are turning the objects themselves into light sources, contributing directly to the scene's illumination.

The Material class in our implementation is designed to manage various material types, including Lambertian, Metal, Dielectric, and DiffuseLight materials. It uses a similar approach to the Hittable class, where we again avoided virtual functions to maintain compatibility with CUDA. The Material class employs an enumeration, ***MaterialType***, to keep track of the material's type. The class also includes a union, ***ObjectUnion***, which can store a pointer to an object of any of the material types. In practice, this union is used to hold the specific material associated with an instance of the Material class.

When a Material object is constructed, its ***type*** field is set to the appropriate value from the ***MaterialType*** enumeration, and the corresponding pointer in the ***ObjectUnion*** is set to point to the specific material object. This structure allows us to use a uniform interface for different materials, simplifying the design of the ray tracer while also facilitating the addition of new material types in the future.

Each material type requires different calculations and considerations, and these variations are key to producing a broad range of visually interesting results. This way, we can create scenes with diverse appearances, contributing to the realism and visual complexity of our ray-traced images.

In Figure 4.4, we provide you with an example of a rendered image from our application showing each of the aforementioned materials. From left to right, we have a dielectric sphere with an index of refraction of 1.5, a metal sphere with 0 fuzziness, and a lambertian sphere with a greenish color. On top of that, we also have a diffuse light sphere to lighten up our scene.

4.11 Textures

Textures play an instrumental role in adding realism and depth to our rendered scenes. They provide a means to add intricate surface detail to our objects, going beyond the use of simple uniform colors. In the context of our ray tracer, we have implemented three kinds of textures: Constant, Checker, and Image textures, each of which is described in more detail in

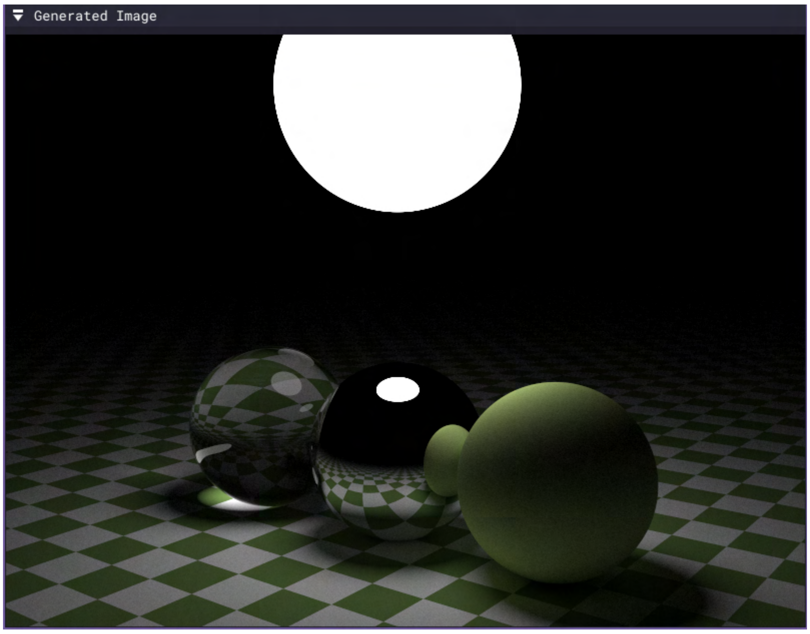


Figure 4.4: Example of Rendered Materials

the following sections.

4.11.1 Constant Textures

Constant textures, as the name suggests, represent a uniform, unchanging color across the entire surface of an object. This kind of texture is one of the simplest to implement and can be used to depict solid colored objects in the scene. According to Shirley's book, a constant texture returns the same color for any given texture coordinates.

In Listing 4.23, we can see an example implementation of the Constant Texture Class.

```
 // Structure for Constant Texture
struct ConstantTexture {
    Vec3 color;  // the constant color of the texture
};

// Value function for Constant Texture
Vec3 value(const float u, const float v, const Vec3& p, const
   ConstantTexture& tex) {
    return tex.color;
}
```

Listing 4.23: Constant Texture

In the Listing above, ***color*** represents the constant color of the texture. The value function always returns this constant color, regardless of the coordinates passed in.

4.11.2 Checker Textures

Checker textures offer a way to create more visually interesting and complex surfaces. This type of texture alternates between two different colors in a checkerboard pattern. To determine the color of a point on the surface, we calculate whether the sum of the integer parts of the texture coordinates is even or odd. If the sum is even, we use the first color; if it's odd, we use the second. Checker textures are especially useful for creating repeating patterns on larger surfaces, such as floors or walls.

In Listing 4.24, the Checker Texture Class is presented.

```
// Structure for Checker Texture
struct CheckerTexture {
    std::shared_ptr<Texture> odd;  // the ”odd” color of the checker pattern
    std::shared_ptr<Texture> even;  // the ”even” color of the checker pattern
};

// Value function for Checker Texture
Vec3 value(const float u, const float v, const Vec3& p, const CheckerTexture& tex) {
    float sines = sin(10*p.x())*sin(10*p.y())*sin(10*p.z());
    if (sines < 0) return tex.odd->value(u, v, p);
    else return tex.even->value(u, v, p);
}
```

Listing 4.24: Checker Texture

For the checker texture, the ***value*** function returns either the ***odd*** or ***even*** color, depending on the result of the sine function evaluated at the point in 3D space.

4.11.3 Image Textures

To further augment the realism of our ray tracing application, we introduced Image Textures, extending the possibilities of surface representation. As opposed to solid textures, which only offer uniform or simple patterned coloration, image textures enable the usage of complex color data from image files, essentially mapping an image onto the surface of objects in our scene.

For image loading and decoding, we relied on the widely-used stb library [28], particularly the ***stb_image.h*** header file. This library supports a broad range of image formats, including JPEG, PNG, TGA, BMP, PSD, GIF, HDR, PIC, and PNM. Notably, its simplistic API and single-file distribution make it an appealing choice for the rapid development and prototyping phase of our ray tracing application.

Once an image is loaded with the STB library, it is treated as an array of pixel values. Each pixel is associated with a color, usually represented as an RGB triplet. To access the color of a specific pixel, we map the texture coordinates ***(u, v)*** to the corresponding pixel in the image.

In essence, the ***value*** function for an Image Texture in our implementation is expressed in the Listing 4.25.

```
// Structure for Image Texture
struct ImageTexture {
    unsigned char* data;  // Pointer to the image data
    int width;            // Width of the image
    int height;           // Height of the image
    int bytes_per_pixel;  // Number of bytes per pixel
    int bytes_per_scanline;  // Number of bytes per scanline
};

// Value function for Image Texture
Vec3 value(float u, float v, const Vec3& p, ImageTexture& tex) const {
    // If no data is present, return a default color.
    if (tex.data == nullptr)
        return Vec3(0, 1, 1);

    // Clamp input texture coordinates to [0,1] x [1,0]
    u = Clamp(u, 0.0f, 1.0f);
    v = 1.0f - Clamp(v, 0.0f, 1.0f); // Flip V to image coordinates

    // Compute the pixel position in the image
    int i = static_cast<int>(u * tex.width);
    int j = static_cast<int>(v * tex.height);

    // Clamp integer mapping, since actual coordinates should be less than 1.0
    if (i >= tex.width)
        i = tex.width - 1;
    if (j >= tex.height)
        j = tex.height - 1;

    // Scale factor to convert color values from [0,255] to [0,1]
    const float color_scale = 1.0f / 255.0f;

    // Calculate the position of the pixel in the image data
    unsigned char* pixel = tex.data + j * tex.bytes_per_scanline + i * tex.bytes_per_pixel;

    // Return the color of the pixel, scaled to [0,1]
    return Vec3(color_scale * pixel[0], color_scale * pixel[1], color_scale * pixel[2]);
}
```

Listing 4.25: Image Texture

The ***value*** function returns the color of an Image Texture at a given point ***(u, v)***.

1. If there's no data available, it returns a default color.

2. It adjusts the texture coordinates ***(u, v)*** to fit within the image dimensions.

3. The texture coordinates are mapped to pixel coordinates in the image.

4. It ensures that the pixel coordinates ***(i, j)*** fall within the image bounds.

5. It calculates a color scale factor to convert color values from [0,255] to [0,1].

6. Finally, it finds the corresponding pixel in the image data and returns its color, scaled to be within the range [0,1].

Our implementation of textures is encapsulated within the Texture class. This class serves as a base class for all texture types. The Texture class is organized similarly to our Hittable and Material classes and adopts a unified interface for all texture types.

The class is constructed with an enumeration, ***TextureType***, defining the types of textures we've implemented: Constant, Checker, and Image. This information is used in the ***ObjectUnion***, which is a union data structure containing pointers to instances of each possible texture type.

In our design, a Texture instance has a ***type*** attribute that defines its texture type, and an ***Object*** attribute, which is a pointer to an ***ObjectUnion***. This ***ObjectUnion*** instance contains a pointer to the specific texture object based on the type attribute. This structure helps us manage different textures in a unified manner, allowing for flexibility and extensibility when implementing various texture types in our ray tracing application.

This encapsulation of texture types within the Texture class streamlines the process of handling different textures, each with their unique value functions. By leveraging polymorphic behavior within a shared interface, we can conveniently use a single texture instance to represent any of the supported texture types.

By employing these three texture types within our ray tracer, we can effectively simulate a wide range of material appearances, greatly enhancing the realism and visual appeal of our rendered scenes. We must note that this level of detail and complexity comes at the cost of increased computational demands, as more intricate texture computations need to be performed for each ray-object intersection. However, the resulting improvement in image quality is well worth the additional computational load.

In Figure 4.5, we provide you with an example of a rendered image from our application, showcasing our University logo in a lambertian, metal and diffuse light rectangle.

4.12 Antialiasing

Antialiasing is a technique used in digital image synthesis to reduce the visual artifacts that occur when high-frequency detail is represented with a low-resolution grid. These artifacts are most noticeable in images as ”jaggies” or stair-stepped edges, which are the result of the grid-like structure of pixel-based displays.

In Ray Tracing, we often use a technique called super-sampling for antialiasing. In this technique, instead of sending a single ray through each pixel, we send multiple rays through different locations within the same pixel, and then average the results. This process smooths out the jaggies because the final color of each pixel is no longer determined by a single point

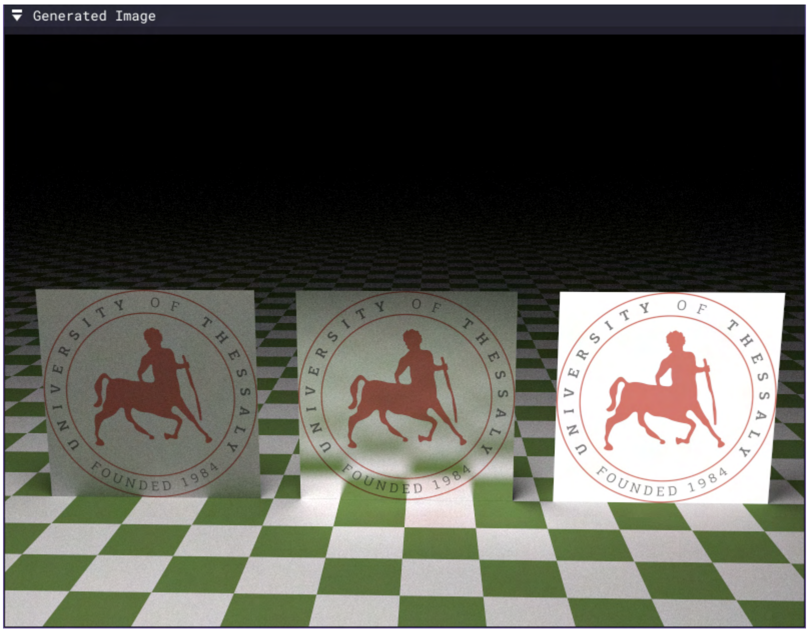


Figure 4.5: Example of Rendered Textures

sample but rather by the average of several samples within that pixel.

Our implementation follows Peter Shirley’s approach as described in ”Ray Tracing in One Weekend” [15]. For each pixel, we generate multiple rays, each slightly offset from the pixel’s center. The number of rays is determined by the variable ***samples_per_pixel***.

In Listing 4.26, we extend the code from 4.7 section to support generating pixels with multiple Samples.

```
for (int s = 0; s < samples_per_pixel; s++) {
    float u = (float)((x - center.x()) + curand_uniform(&local_rand_state)) / (float)(width);
    float v = (float)((center.y() - y) + curand_uniform(&local_rand_state)) / (float)(width);
    Vec3 distFromCenter = (u * rightV) + (v * upV);
    Vec3 startPos = (inputs.near_plane * distFromCenter) + origin + (inputs.fov * forwardV);
    Vec3 secondPlanePos = (inputs.far_plane * distFromCenter) + ((1.0f / inputs.fov * 10.0f) * forwardV) + origin;
    Vec3 dirVector = Normalize(secondPlanePos - startPos);

    Ray r = Ray(startPos, dirVector);
    col = col + color(r, world, max_depth, &local_rand_state, inputs);
}
```

Listing 4.26: Antialiasing

Each sample is sent through a slightly different part of the pixel, with the offsets ***u*** and ***v*** determined by random factors. The function ***curand_uniform*** generates a random number between 0 and 1, which is used to offset the sample within the pixel.

This process effectively averages the color over the area of the pixel, instead of determining the color based on the center of the pixel alone. This technique reduces the jagged or stair-stepped effect that can occur when the scene is sampled at the pixel resolution. The more samples per pixel, the more accurate the color representation and the smoother the resulting image will appear.

In Figure 4.6, is an example of before and after antialiasing in our application. We can clearly observe the difference of the pixels at the edge of the sphere.

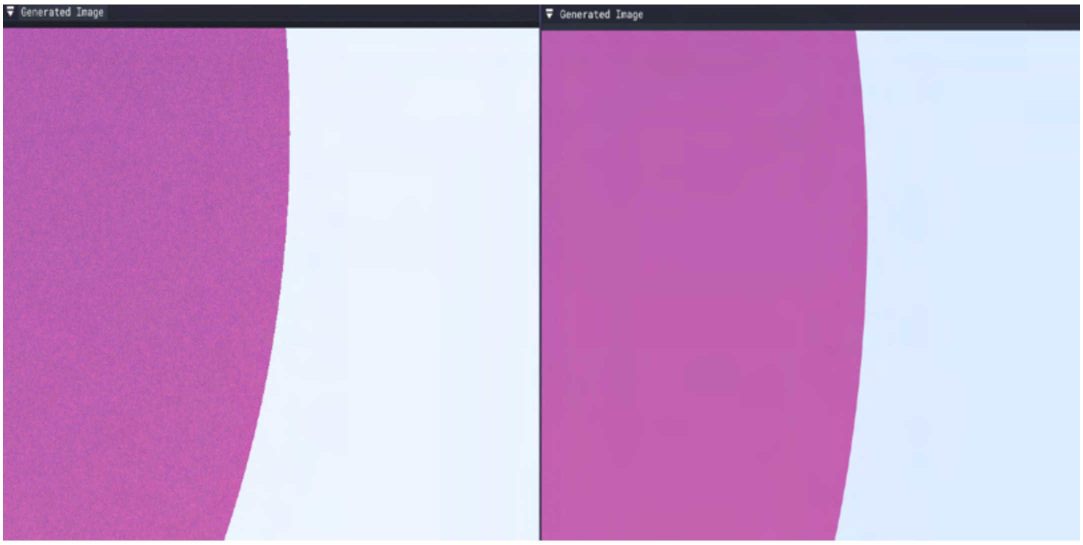


Figure 4.6: Before and after Antialiasing

4.13 Bounding Volume Hierarchies

Bounding Volume Hierarchies (BVHs) [2] are a crucial optimization technique used in ray tracing to reduce the number of intersection tests. By intelligently grouping objects into hierarchies of bounding volumes, we can often eliminate many unnecessary intersection tests, which can significantly speed up the rendering process.

The idea behind a BVH is simple: instead of testing each ray against every object in the scene, we first test the ray against a simple bounding volume containing a group of objects. If the ray doesn't intersect the bounding volume, we can immediately discard all the objects inside it without testing them individually. If the ray does intersect the bounding volume, we then need to test the ray against each object contained within that volume.

BVHs are typically implemented as binary trees. Each node in the tree contains a bounding volume and two children. The children can be either inner nodes (with their bounding volumes) or leaf nodes, representing actual geometric objects in the scene. The tree is constructed so that each bounding volume completely encloses the bounding volumes and objects of its children.

In our implementation, we followed the approach described by Peter Shirley in ”Ray Tracing: The Next Week” [39] and modified it to our needs. We construct a BVH by sorting the scene objects by their geometrical type and then recursively splitting them into two groups. The objects are enclosed within an axis-aligned bounding box (AABB), which makes the intersection test with the ray very fast and straightforward. In Listing 4.27, is a high-level view of how this might look in code.

```
class BVHNode {
public:
    Hittable* left;
    Hittable* right;
    AABB box;

    BVHNode(Hittable** list, size_t start, size_t end)
    {
        // ...
        // Construct the BVH node here by sorting objects by their type and splitting them into two groups
        // ...
    }

    inline bool Hit(const Ray& r, float t_min, float t_max, HitRecord& rec) const
    {
        if (!box.hit(r, t_min, t_max))
            return false;

        bool hit_left = left->hit(r, t_min, t_max, rec);
        bool hit_right = right ? right->hit(r, hit_left ? rec.t : t_max,
    rec) : false;

        return hit_left || hit_right;
    }
};
```

Listing 4.27: BVHNode Class

When a ray is tested against a BVH node, it first tests against the node's bounding box. If there's no intersection, it's clear the ray doesn't hit any object in the subtree, and the function returns false. If it hits, the ray tests against the children of the node, which recursively applies the same logic until it reaches the leaf nodes representing actual scene objects. This recursive tree traversal is efficient and allows for rapid culling of irrelevant objects, significantly speeding up the ray tracing process. Please note that in our actual implementation, we used an iterative approach, since recursion in CUDA is not recommended due to its high stack usage and performance issues.

4.14 CUDA Implementation Specifics

When we set out to implement our ray tracer, a key goal was to take advantage of the high performance parallel computing capabilities offered by NVIDIA's CUDA. As we described in section 3.2, this necessitated several design decisions in terms of memory management, computational strategy, and more, which were essential to optimize our ray tracing algorithm and ensure efficient execution on the GPU. In this section, we discuss the specifics of our CUDA implementation, describing how we've exploited certain characteristics of CUDA to enhance our application's performance.

4.14.1 Kernel Configuration

Kernel configuration is an integral aspect of our CUDA implementation strategy, requiring careful tuning to strike a balance between performance and resource consumption. An essential part of this process is the configuration of kernel launches, where we have to specify the number of thread blocks and the number of threads per block for each kernel invocation.

In our case, we've settled on using 1024 threads per kernel, guided by the hardware capabilities of modern GPU architectures, which commonly support up to 1024 threads per block. This number, while being a good starting point, is not set in stone. Depending on the specific GPU architecture in use and the nature of the problem at hand, different configurations might yield better performance [11].

The ***__launch_bounds__*** directive has proven to be an indispensable tool in this process. By using this directive, we're able to provide the CUDA compiler with an upper bound on the number of threads per block for a specific kernel. This information enables the compiler to make better decisions on how to allocate and optimize the use of the GPU's scarce and valuable resources, such as registers and shared memory. In our implementation, the ***__launch_bounds__*** directive has been set with the maximum thread block size of 1024, aligning with our chosen configuration.

Profiling our application with NVIDIA's Nsight Compute [20], the occupancy of our application seems optimal since the theoretical occupancy of our application is at 100%. As for, the achieved occupancy of our application is at 83.3%. The difference between calculated theoretical (100.0%) and measured achieved occupancy (83.3%) can be the result of warp scheduling overheads or workload imbalances during the kernel execution. Occupancy in this context is the percentage of the hardware's ability to process warps that is actively in use. Warps are threads that are automatically grouped into bundles [40]. The number of threads in a warp is a bit arbitrary, and it depends on the manufacturer of the chip.

In Figure 4.7, we can visualize the occupancy graphs of our application. The blue dot on the graphs represents our chosen resource usage of the kernel. The other data points represent the range of possible block sizes, register counts, and shared memory allocation.

Please note that higher occupancy does not always result in higher performance, however, low occupancy always reduces the ability to hide latencies, resulting in overall performance degradation. Large discrepancies between the theoretical and the achieved occupancy during execution typically indicates highly imbalanced workloads.

4.14.2 Coalesced Memory Access

Performance optimization in CUDA goes beyond just configuring the kernel; it extends to memory management and access patterns as well. A key strategy we employed in our application is Coalesced Memory Access, leveraging the function ***cudaMallocManaged*** to

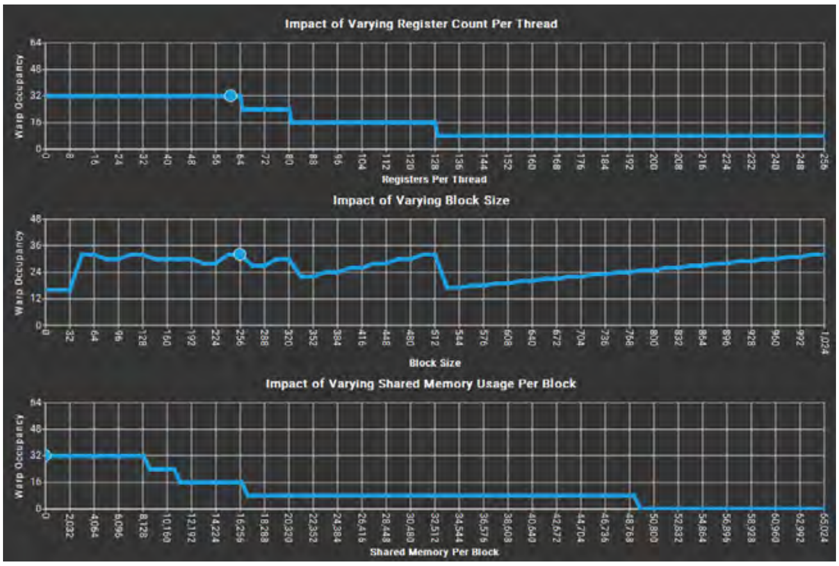


Figure 4.7: NVIDIA Nsight Compute Occupancy Graphs

allocate Unified Memory, a shared memory space accessible by both CPU and GPU [41].

Coalesced Memory Access refers to the process whereby consecutive threads access consecutive memory locations. This pattern is highly beneficial as it maximizes memory bandwidth utilization by allowing the threads to fetch memory in a single transaction, thus greatly reducing the number of transactions required. As GPUs are designed to access memory in large, aligned chunks, coalescing memory accesses ensures that the full bandwidth of the memory subsystem is harnessed, leading to efficient memory use and improved performance.

Furthermore, we did some padding on our data structures when their sizes were not multiples of the warp size (32 bytes). Padding our data structures, helped us ensure that memory accesses by threads in different warps do not fall into the same cache line, which can improve coalescing.

In the following steps, we will explain how we implemented global memory coalescing in our application:

1. Define the size of each type of Object (Hittables, Materials, Textures)

2. Calculate the total size of memory needed

3. Allocate unified memory with ***cudaMallocManaged*** function

4. Partition memory for the scene’s object and their constituent objects

5. Initialize the hittables with the proper constructors for the material and texture

In addition to adopting coalesced memory access, we employed the advanced features of NVIDIA's Nsight Compute, including the Memory Workload Analysis tool and Source Counters. The Memory Workload Analysis tool was instrumental in examining our application's memory access patterns, identifying inefficiencies, and ensuring that we were achieving the desired coalesced access.

Source Counters, on the other hand, offered us granular insights into the GPU's warp execution efficiency and allowed us to analyze the behavior of our code at the instruction level. With this feature, we could directly correlate memory access patterns with specific code lines, helping us better understand where inefficiencies occurred and how we could optimize them.

With the graphical and detailed breakdown provided by these tools, we identified uncoalesced accesses, out-of-bounds accesses, and other potential issues. Then, we addressed them promptly, leading to substantial improvements in our ray tracer's performance. This iterative process of analysis, modification, and verification, augmented by Nsight Compute's comprehensive profiling capabilities, played a crucial role in optimizing our memory access patterns, thereby enhancing the overall performance of our application.

4.14.3 Iteration over Recursion

Recursion is an important algorithmic technique often used in programming, but in the context of CUDA and GPU programming, it comes with some specific limitations and potential pitfalls. The stack size on a GPU is limited compared to a CPU, making it more prone to stack overflow if recursion depth becomes too large. Moreover, each thread in CUDA has its own private stack. This means, when recursion is used extensively, it can lead to substantial memory overhead. Recursion can also hinder performance due to the absence of tail-call optimization [42] in CUDA, potentially leading to inefficient use of the GPU's resources.

Given these factors, we consciously chose to use iteration instead of recursion in our device code to circumvent these challenges [43]. This choice was particularly impactful when applied to our ray-color computation, where we transformed the recursive function into a loop using a manual stack.

An example implementation of the ***color*** device function can be seen in Listing 4.28.

```
__device__ inline Vec3 color(const Ray& r, Hittable* world, int max_depth
    )
```

```
{
    Ray cur_ray = r;
    Vec3 cur_attenuation = Vec3(1.0f, 1.0f, 1.0f);
    Vec3 background = Vec3(0.0f, 0.0f, 0.0f);

    HitRecord rec;

    for (int i = 0; i < max_depth; i++) {
        if (!Hit(cur_ray, 0.001f, FLT_MAX, rec)) {
            // return the background color (sky)
        }
        else {
            Vec3 emitted = Vec3(0.0f, 0.0f, 0.0f);
            Ray scattered;
            Vec3 attenuation;

            switch (material->type) {
            // return calculated color based on material type.
            // color = emmited * cur_attenuation
            default:
                return background;
            }

            cur_attenuation = attenuation * cur_attenuation;
            cur_ray = scattered;
        }
    }

    return background; // exceeded recursion
}
```

Listing 4.28: Color Computation Device Function

In the BVHNode's Hit device function, we faced a similar situation. The BVHNode represents a Bounding Volume Hierarchy (BVH), a tree structure used in ray tracing to improve rendering speed 4.13. Each BVHNode can contain two child nodes, hence the traversal through the BVH traditionally uses a recursive approach. However, following our strategy, we reworked this to use iteration. This iterative solution uses a manual stack to store the nodes for traversal, removing the risk of stack overflow and improving the performance consistency.

Our switch from recursion to iteration allowed us to better manage GPU memory, reduce latency, and increase throughput. It also led to more predictable performance, as it reduced the variance in execution times across threads that is usually introduced by the non-uniform recursion depths. This change, coupled with our other optimization efforts, contributed to significant performance improvement in our ray tracing application.

4.14.4 Avoiding Virtual Functions

The CUDA programming model provides support for class inheritance and virtual functions. However, in practice, the use of virtual functions in CUDA device code can lead to performance degradation [44]. This arises due to two primary reasons: an increase in instruction divergence and the requirement of slower global memory access for the virtual function table, or vtable.

Instruction divergence occurs when threads within the same warp follow different execution paths. Since the GPU is a SIMD (Single Instruction, Multiple Data) architecture, it is most efficient when all threads in a warp perform the same operation. In the context of virtual functions, this divergence arises because different objects may execute different function implementations depending on their dynamic type, leading to inconsistent execution paths among threads.

Further, the vtable, which is used for dispatching calls to the correct function implementation, typically resides in global memory. Global memory accesses are much slower than shared or local memory accesses, leading to increased latency.

In order to mitigate these issues, we employed a design pattern that involves the use of unions in our core classes, such as the Hittable, Material, and Texture. By using unions, we were able to store different types of data in the same memory location and determine at runtime which variant to use, without the need for virtual functions. This approach enabled us to effectively emulate the polymorphic behavior usually provided by virtual functions, while avoiding the associated performance costs.

Furthermore, by structuring our data in this way, we were able to keep related data close together in memory, improving memory access patterns and potentially leveraging the hardware cache more effectively. This change was instrumental in the optimization of our ray tracer, demonstrating that careful consideration of hardware characteristics and CUDA-specific features can significantly influence application performance.

4.14.5 Float Precision

Our ray tracer primarily uses single-precision (32-bit) floats in calculations to maintain efficient computation and maximize performance. This decision was based on the observation that GPUs, including those we've employed for our study, typically exhibit a higher throughput for single-precision floating-point operations as compared to double-precision operations. This is a design characteristic inherent to many contemporary GPUs, stemming from their original role as processors for graphics rendering, a task where single-precision computations are often sufficient.

The use of single-precision computations aligns with the architectural strengths of the GPU, allowing us to fully leverage its computational capabilities. The enhanced speed of single-precision operations can significantly reduce the overall computation time of our ray tracing algorithm, thereby enabling the processing of complex scenes in a reasonable time frame.

In addition to using single-precision floats, we have made extensive use of CUDA's intrinsic functions for arithmetic operations in our ray tracer. These intrinsic functions are specially designed to exploit the GPU's hardware capabilities for specific mathematical operations, resulting in faster and potentially more accurate computations. They provide an additional layer of optimization, and their use can lead to noticeable performance improvements in the ray tracer.

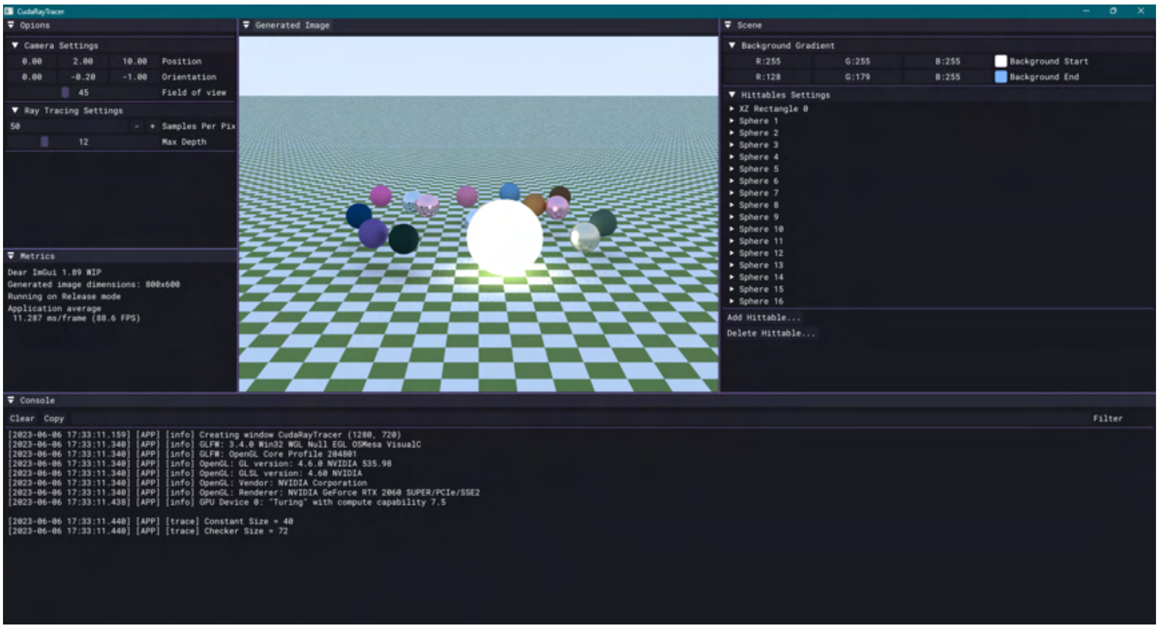
However, it's crucial to mention that the usage of single-precision floats also requires careful consideration. Although they offer substantial performance benefits, they come with lower precision compared to double-precision floats. This can introduce errors in calculations, especially for larger or more complex computations. Therefore, the choice of precision in ray tracing, or in any application, requires a careful balance between performance and the level of precision required by the task at hand. In our work, we found that single-precision was adequate for our needs, without introducing noticeable inaccuracies in our rendered images.

In conclusion, the specific techniques we have employed for our CUDA implementation are critical in leveraging the parallel processing capabilities of modern GPUs, and have significantly contributed to the performance of our ray tracer. These techniques demonstrate that a comprehensive understanding of the CUDA architecture, coupled with an effective problem-solving approach, is key to efficient GPU programming.

4.15 User Interaction and Experience

Our ray tracing application offers an engaging and interactive user experience, built around a feature-rich graphical user interface developed with ImGui [3]. The UI is designed to be intuitive and responsive, ensuring that users, whether they are beginners or experienced with ray tracing, can easily interact with the application, modify settings, and visualize the effects of their adjustments in real-time.

In Figure 4.8, is an overview of our application's complete UI.



Camera Settings

0.00 2.00 10.00 Position
0.00 -0.20 -1.00 Orientation
45 Field of view

Ray Tracing Settings

50 - + Samples Per Pix
12 Max Depth

Dear ImGui 1.89 WIP
Generated image dimensions: 800x600
Running on Release mode
Application average
11.287 ms/frame (88.6 FPS)

Background Gradient

R:255 G:255 B:255 Background Start
R:128 G:179 B:255 Background End

Hittables Settings

- XZ Rectangle 0
- Sphere 1
- Sphere 2
- Sphere 3
- Sphere 4
- Sphere 5
- Sphere 6
- Sphere 7
- Sphere 8
- Sphere 9
- Sphere 10
- Sphere 11
- Sphere 12
- Sphere 13
- Sphere 14
- Sphere 15
- Sphere 16

```
[2023-06-06 17:33:11.159] [APP] [info] Creating window CudaRayTracer (1280, 720)
[2023-06-06 17:33:11.340] [APP] [info] GLFW: 3.4.0 Win32 WGL Null EGL OSMesa VisualC
[2023-06-06 17:33:11.340] [APP] [info] GLFW: OpenGL Core Profile 204801
[2023-06-06 17:33:11.340] [APP] [info] OpenGL: GL version: 4.6.0 NVIDIA 535.98
[2023-06-06 17:33:11.340] [APP] [info] OpenGL: GLSL version: 4.60 NVIDIA
[2023-06-06 17:33:11.340] [APP] [info] OpenGL: Vendor: NVIDIA Corporation
[2023-06-06 17:33:11.340] [APP] [info] OpenGL: Renderer: NVIDIA GeForce RTX 2060 SUPER/PCIe/SSE2
[2023-06-06 17:33:11.438] [APP] [info] GPU Device 0: "Turing" with compute capability 7.5

[2023-06-06 17:33:11.440] [APP] [trace] Constant Size = 40
[2023-06-06 17:33:11.440] [APP] [trace] Checker Size = 72
```

Figure 4.8: Application's UI

4.15.1 CUDA Generated Image Window

The CUDA Image Window is where the ray tracing magic happens. This window displays the rendered scene in real-time as rays are traced through the scene, bouncing off objects, and accumulating color information. It serves as a canvas that dynamically updates to reflect any changes made through the other interface elements, showcasing the power and flexibility of ray tracing in producing visually stunning images.

In Figure 4.9, we can see the Generated Image ImGui window.

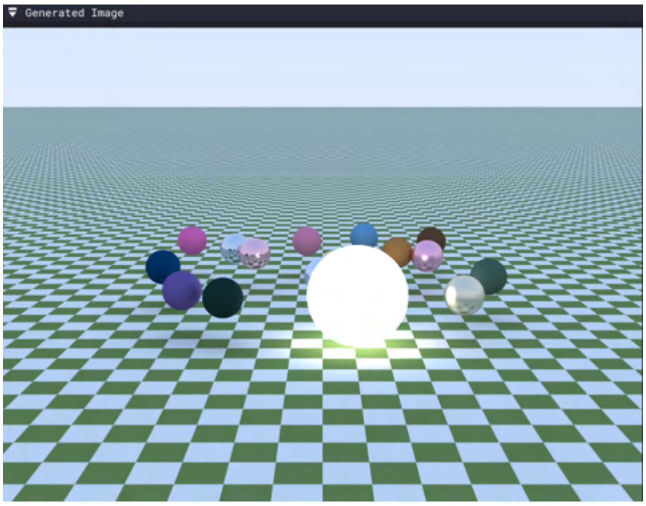
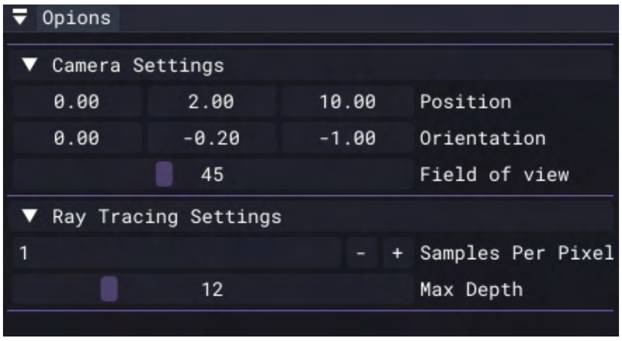


Figure 4.9: Generated Image Window

4.15.2 Options Window

The Options Window provides users with granular control over camera and ray tracing settings. The adjustable camera settings include field of view, camera position and orientation, allowing the user to view the scene from different perspectives. The ray tracing settings include modifying the maximum depth of ray bounces and adjusting the number of samples per pixel that we described in 4.12 section. These controls provide users with the ability to fine-tune the ray tracing process to meet their specific needs and preferences.

In Figure 4.10, we can see the Options window with the adjustable Camera and Ray Tracing settings.



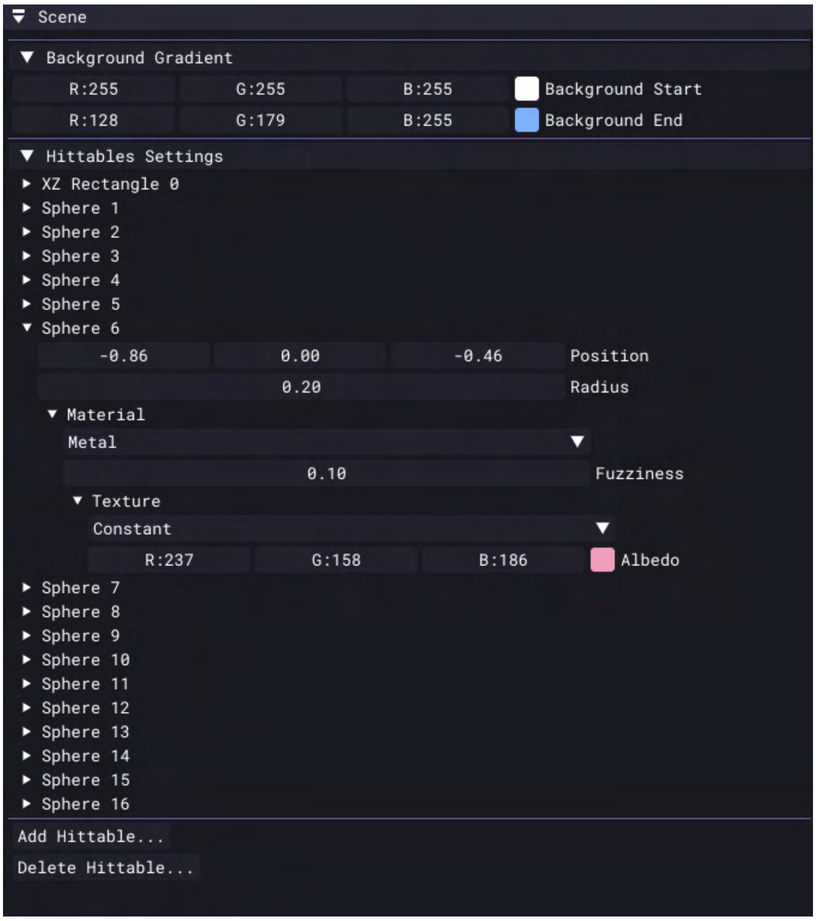
```
Opions
▼ Camera Settings
0.00    2.00    10.00   Position
0.00   -0.20    -1.00   Orientation
        45              Field of view
▼ Ray Tracing Settings
1                 - +   Samples Per Pixel
        12              Max Depth
```

Figure 4.10: Options Window

4.15.3 Scene Window

The Scene Window is where users can interact directly with the objects in the scene. Users can add or remove objects, adjust their position and size, and modify the scene's objects materials and textures. This includes changing the color, reflectivity, and texture of objects. The Scene Window also allows users to adjust the background color of the scene, offering another dimension of customization.

In Figure 4.11, is the Scene window with the adjustable Background and Objects settings. For example, if we want to change the ***Sphere 6***'s material or texture, we can do so by clicking on the dropdown button, respectively.



Scene

▼ Background Gradient

R:255 G:255 B:255 ⬜ Background Start

R:128 G:179 B:255 🟦 Background End

▼ Hittables Settings

- ▶ XZ Rectangle 0
- ▶ Sphere 1
- ▶ Sphere 2
- ▶ Sphere 3
- ▶ Sphere 4
- ▶ Sphere 5
- ▼ Sphere 6

  -0.86 0.00 -0.46 Position

  0.20 Radius

  ▼ Material

  Metal ▼

  0.10 Fuzziness

  ▼ Texture

  Constant ▼

  R:237 G:158 B:186 🟪 Albedo

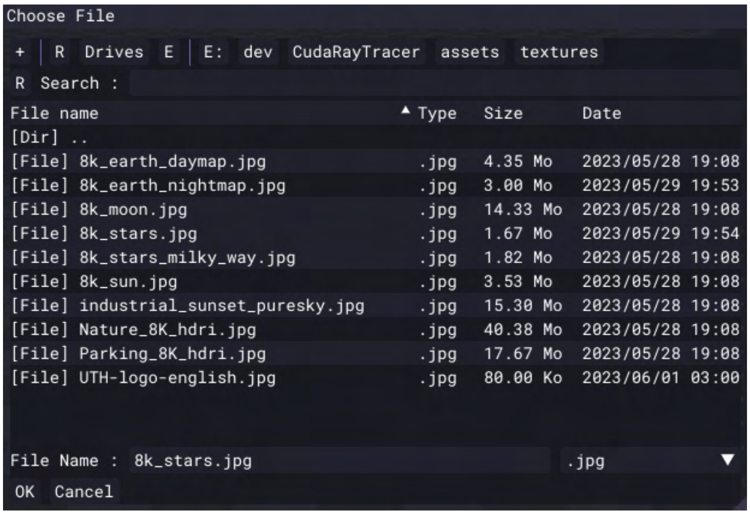
- ▶ Sphere 7
- ▶ Sphere 8
- ▶ Sphere 9
- ▶ Sphere 10
- ▶ Sphere 11
- ▶ Sphere 12
- ▶ Sphere 13
- ▶ Sphere 14
- ▶ Sphere 15
- ▶ Sphere 16

Add Hittable...

Delete Hittable...

Figure 4.11: Scene Window

Furthermore, if the type of texture on an object is an ***Image***, then an ***Open...*** button appears. Clicking on the button, a popup file dialog window appears for choosing the desired image file, as seen in Figure 4.12.



File name	Type	Size	Date
[Dir] ..			
[File] 8k_earth_daymap.jpg	.jpg	4.35 Mo	2023/05/28 19:08
[File] 8k_earth_nightmap.jpg	.jpg	3.00 Mo	2023/05/29 19:53
[File] 8k_moon.jpg	.jpg	14.33 Mo	2023/05/28 19:08
[File] 8k_stars.jpg	.jpg	1.67 Mo	2023/05/29 19:54
[File] 8k_stars_milky_way.jpg	.jpg	1.82 Mo	2023/05/28 19:08
[File] 8k_sun.jpg	.jpg	3.53 Mo	2023/05/28 19:08
[File] industrial_sunset_puresky.jpg	.jpg	15.30 Mo	2023/05/28 19:08
[File] Nature_8K_hdri.jpg	.jpg	40.38 Mo	2023/05/28 19:08
[File] Parking_8K_hdri.jpg	.jpg	17.67 Mo	2023/05/28 19:08
[File] UTH-logo-english.jpg	.jpg	80.00 Ko	2023/06/01 03:00

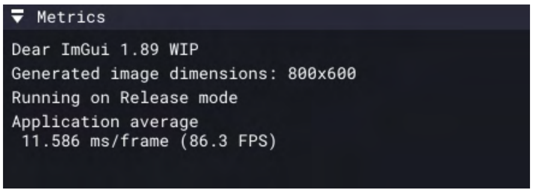
File Name : 8k_stars.jpg .jpg ▼

Figure 4.12: File Dialog Window

4.15.4 Metrics Window

The Metrics Window serves as a real-time performance dashboard, providing information about the rendering speed of the application, the running mode of the application (Debug, Release, Dist), the generated image dimensions, and the time taken to render the current frame. These metrics are invaluable for users interested in optimizing their scenes for performance or comparing the speed of different hardware configurations.

In Figure 4.13, is the Metrics Window.



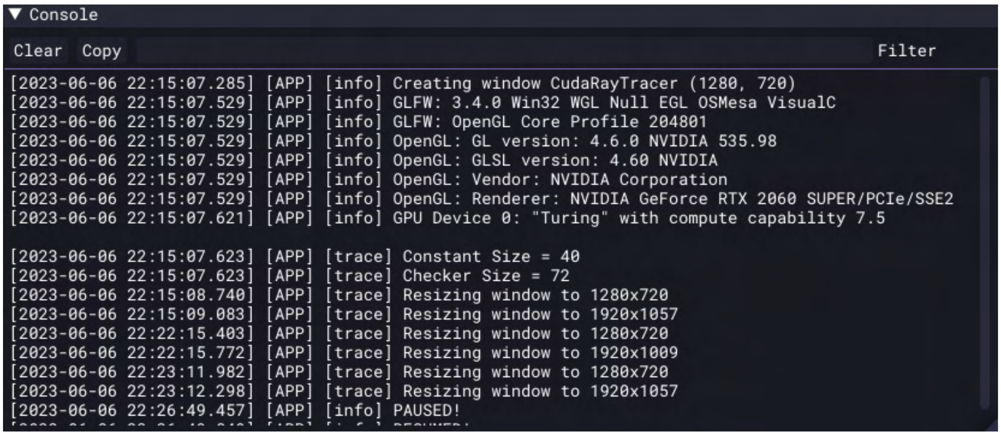
```
Dear ImGui 1.89 WIP
Generated image dimensions: 800x600
Running on Release mode
Application average
 11.586 ms/frame (86.3 FPS)
```

Figure 4.13: Metrics Window

4.15.5 Console Window

The Console Window is the hub of application log information. It displays important events, error messages, and status updates from the application, keeping users informed about the application's operations and helping them diagnose any issues that might arise.

In Figure 4.14, is the Console Window. We can also filter the logs with the searching bar on top of the window, clear the whole log, and copy it.



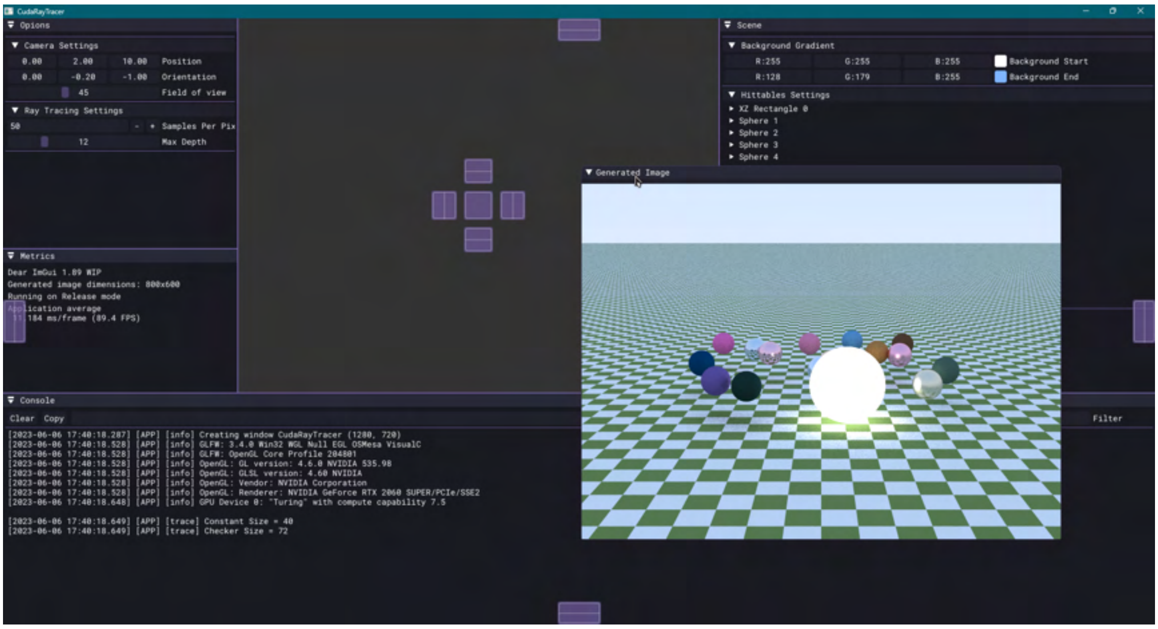
```
[2023-06-06 22:15:07.285] [APP] [info] Creating window CudaRayTracer (1280, 720)
[2023-06-06 22:15:07.529] [APP] [info] GLFW: 3.4.0 Win32 WGL Null EGL OSMesa VisualC
[2023-06-06 22:15:07.529] [APP] [info] GLFW: OpenGL Core Profile 204801
[2023-06-06 22:15:07.529] [APP] [info] OpenGL: GL version: 4.6.0 NVIDIA 535.98
[2023-06-06 22:15:07.529] [APP] [info] OpenGL: GLSL version: 4.60 NVIDIA
[2023-06-06 22:15:07.529] [APP] [info] OpenGL: Vendor: NVIDIA Corporation
[2023-06-06 22:15:07.529] [APP] [info] OpenGL: Renderer: NVIDIA GeForce RTX 2060 SUPER/PCIe/SSE2
[2023-06-06 22:15:07.621] [APP] [info] GPU Device 0: "Turing" with compute capability 7.5

[2023-06-06 22:15:07.623] [APP] [trace] Constant Size = 40
[2023-06-06 22:15:07.623] [APP] [trace] Checker Size = 72
[2023-06-06 22:15:08.740] [APP] [trace] Resizing window to 1280x720
[2023-06-06 22:15:09.083] [APP] [trace] Resizing window to 1920x1057
[2023-06-06 22:22:15.403] [APP] [trace] Resizing window to 1280x720
[2023-06-06 22:22:15.772] [APP] [trace] Resizing window to 1920x1009
[2023-06-06 22:23:11.982] [APP] [trace] Resizing window to 1280x720
[2023-06-06 22:23:12.298] [APP] [trace] Resizing window to 1920x1057
[2023-06-06 22:26:49.457] [APP] [info] PAUSED!
```

Figure 4.14: Console Window

4.15.6 Dockable Interface

Thanks to the use of the ImGui docking branch, all the above windows can be freely repositioned, resized, and docked according to the user's preference. This means users have full control over the layout of the interface, enabling them to create a workspace that best suits their workflow.

Figure 4.15 showcases how the ImGui docking interface works, by dragging an ImGui Window.

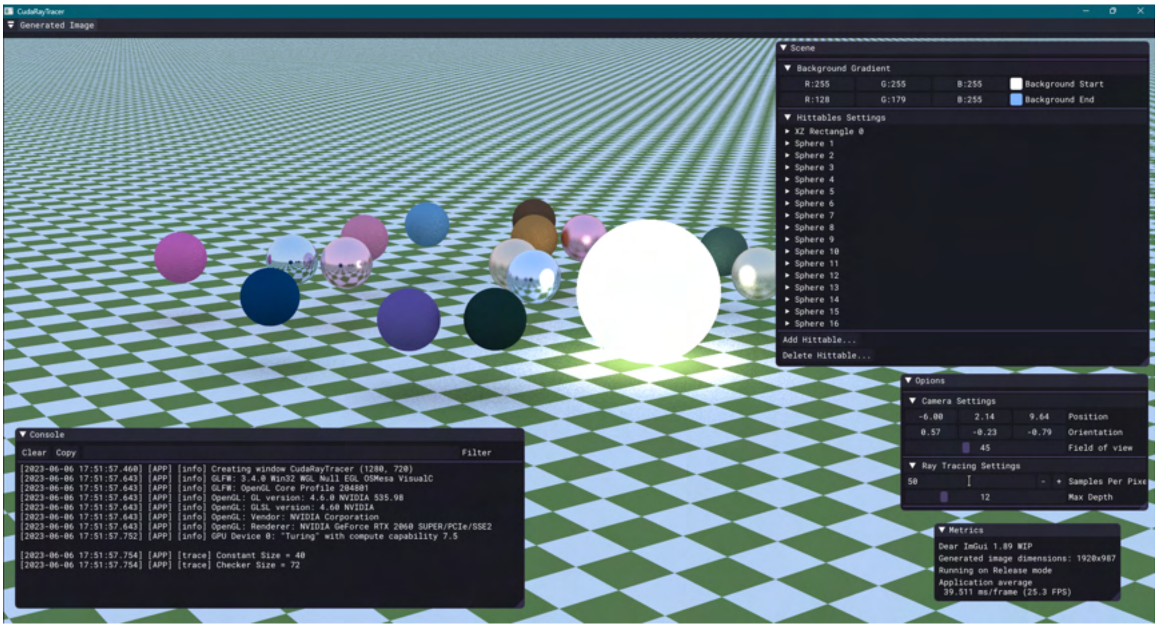


```
0.00   2.00   10.00  Position
0.00   -0.20  -1.00  Orientation
       45            Field of view
50            - +    Samples Per Pix
       12            Max Depth

R:255   G:255   B:255   Background Start
R:128   G:179   B:255   Background End

Dear ImGui 1.89 WIP
Generated image dimensions: 800x600
Running on Release mode
Application average
11.184 ms/frame (89.4 FPS)
```

```
[2023-06-06 17:40:18.287] [APP] [info] Creating window CudaRayTracer (1280, 720)
[2023-06-06 17:40:18.528] [APP] [info] GLFW: 3.4.0 Win32 WGL Null EGL OSMesa VisualC
[2023-06-06 17:40:18.528] [APP] [info] GLFW: OpenGL Core Profile 204801
[2023-06-06 17:40:18.528] [APP] [info] OpenGL: GL version: 4.6.0 NVIDIA 535.98
[2023-06-06 17:40:18.528] [APP] [info] OpenGL: GLSL version: 4.60 NVIDIA
[2023-06-06 17:40:18.528] [APP] [info] OpenGL: Vendor: NVIDIA Corporation
[2023-06-06 17:40:18.528] [APP] [info] OpenGL: Renderer: NVIDIA GeForce RTX 2060 SUPER/PCIe/SSE2
[2023-06-06 17:40:18.648] [APP] [info] GPU Device 0: "Turing" with compute capability 7.5

[2023-06-06 17:40:18.649] [APP] [trace] Constant Size = 40
[2023-06-06 17:40:18.649] [APP] [trace] Checker Size = 72
```

Figure 4.15: ImGui Docking

In Figure 4.16, is an alternative ImGui customization.



```
R:255    G:255    B:255    Background Start
R:128    G:179    B:255    Background End
```

```
-6.00    2.14     9.64     Position
0.57     -0.23    -0.79    Orientation
         45                Field of view
50                 - +     Samples Per Pixe
         12                Max Depth
```

```
[2023-06-06 17:51:57.460] [APP] [info] Creating window CudaRayTracer (1280, 720)
[2023-06-06 17:51:57.643] [APP] [info] GLFW: 3.4.0 Win32 WGL Null EGL OSMesa VisualC
[2023-06-06 17:51:57.643] [APP] [info] GLFW: OpenGL Core Profile 204801
[2023-06-06 17:51:57.643] [APP] [info] OpenGL: GL version: 4.6.0 NVIDIA 535.98
[2023-06-06 17:51:57.643] [APP] [info] OpenGL: GLSL version: 4.60 NVIDIA
[2023-06-06 17:51:57.643] [APP] [info] OpenGL: Vendor: NVIDIA Corporation
[2023-06-06 17:51:57.643] [APP] [info] OpenGL: Renderer: NVIDIA GeForce RTX 2060 SUPER/PCIe/SSE2
[2023-06-06 17:51:57.752] [APP] [info] GPU Device 0: "Turing" with compute capability 7.5

[2023-06-06 17:51:57.754] [APP] [trace] Constant Size = 40
[2023-06-06 17:51:57.754] [APP] [trace] Checker Size = 72
```

```
Dear ImGui 1.89 WIP
Generated image dimensions: 1920x987
Running on Release mode
Application average
 39.511 ms/frame (25.3 FPS)
```

Figure 4.16: ImGui Alternative Customization

4.15.7 Keyboard Controls for User Input

In addition to the graphical interface, our application also incorporates keyboard controls to facilitate camera navigation and certain functionality. Once the user clicks inside the Generated Image Window to gain focus, they can utilize the following keys:

- ***W, A, S, D***: These keys allow the user to control the camera position, akin to common navigation controls in video games. ***W*** and ***S*** move the camera forward and backward along the viewing direction, respectively, while ***A*** and ***D*** shift the camera left and right, perpendicular to the viewing direction.

- ***Ctrl and Space***: To adjust the vertical position of the camera, users can use the ***Ctrl*** key to move downward and the ***Space*** key to move upward. This feature allows users to explore the scene from different vertical perspectives and positions.

- ***Shift***: To adjust the speed of the camera, users can hold the ***Shift*** key to gain velocity inside the scene. This feature is particularly useful when the user wants to move the camera a great distance.

- ***P***: This key serves a unique function - it allows the user to pause and resume the ray tracing process. This functionality can be particularly useful when the user wishes to halt the computation to examine the current state of the scene or to make adjustments in the settings before resuming the rendering process.

These keyboard controls provide a more immersive and interactive user experience, giving users an additional layer of control over the viewing and rendering process. They exemplify our application's commitment to a user-friendly, interactive, and responsive interface for exploring the possibilities of ray tracing.

Overall, our application's interface is not just a tool for ray tracing; it's an immersive environment that puts the user in the driver's seat, enabling them to unleash their creativity, experiment with ray tracing parameters, and witness the impact of their changes in real-time.

Chapter 5

Experiments, Conclusions and Future Work

As we draw our exploration of GPU-accelerated ray tracing to a close, this final chapter serves to wrap up our research, review our accomplishments, and consider the potential for further developments. We will take a moment to reflect on the journey that has led us to successfully build a CUDA-based real-time ray tracer, contemplating the challenges we encountered and the solutions we devised. We will also delve into the testing and experimental evaluation of our application, analyzing its performance and comparing it with CPU-based implementations. This empirical examination will reinforce our conclusions and shed light on the benefits and drawbacks of our approach. Lastly, we will look forward into the future, discussing existing limitations of our ray tracer, outlining potential bug fixes, and envisioning additional features and enhancements. Through this final chapter, we will encapsulate the breadth of our work, providing a comprehensive overview of our findings and speculating on the exciting paths that lie ahead in GPU-accelerated ray tracing.

5.1 Experiments and Testing

In order to validate the effectiveness of our GPU-accelerated ray tracing application and compare its performance against other versions of Ray Tracing in One Weekend 3, we conducted a series of tests on the three different implementations: RTOW running on a CPU, RTOW running on a CUDA-enabled GPU, and our own CUDA-based application.

The tests involved rendering the same scene across all three versions with varying input sizes, represented by the number of samples per pixel (SPP). The SPP values chosen for these tests were 1, 10, 50, 100, 500, and 1000. The scene used for these tests was kept consistent to ensure a fair comparison. The scene used for testing is displayed in Figure 5.1. The scene consists of 100 spheres arranged in a 10x10 grid pattern, with positions offset slightly by a random factor for variation and realism. Each sphere in this scene has a radius of 0.2 units and is centered at a position determined by its grid coordinates, again adjusted by a random factor. The materials of the spheres -—Lambertian, Metal, or Dielectric—- are chosen randomly, with respective probabilities of 0.8, 0.15, and 0.05. The material characteristics are as follows:

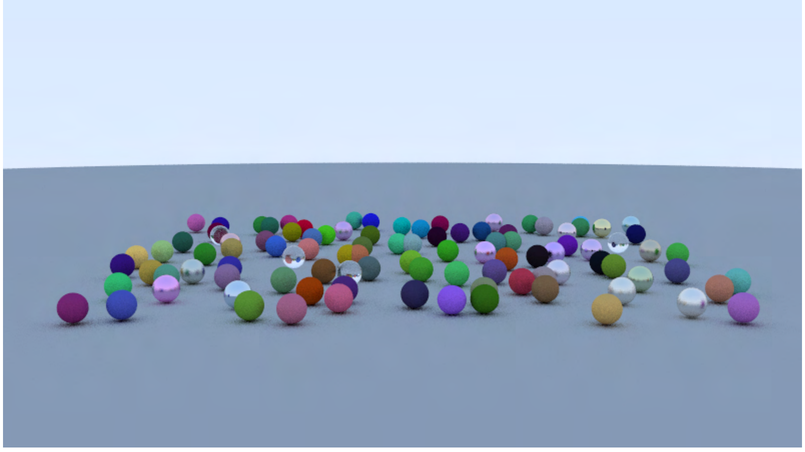


Figure 5.1: Testing Scene

1. Lambertian spheres: These spheres are diffuse, with their colors determined by the square of a random RGB value. This method of color assignment ensures that darker colors are more frequently chosen, emulating the common real-world observation that darker colors are generally more prevalent.

2. Metal spheres: These spheres have a reflective surface, with their colors determined by an RGB value that is an average of a random value and 0.5, ensuring that they have a brighter, more metallic look. Additionally, these spheres have a fuzz factor, which is a value up to 0.5 that determines the amount of random reflection.

3. Dielectric spheres: These spheres are transparent and refract light, simulating materials like glass or water. The refractive index is set at 1.5, typical of glass.

The randomness in color, position, and material type contributes to the complexity of the scene, making it an effective testbed for our ray tracer’s capabilities and performance.

To gain a robust measure of each version's execution time, we ran each test five times and computed the average execution time in milliseconds. Averaging the results over multiple runs helps to reduce the impact of random variations and provide a more accurate measure of performance.

The execution time for each version was obtained by running the executables with a Python script that utilized the subprocess module. This script not only ran the executables, but also collected the output and calculated the average execution times.

The results of these tests are visualized in a graph in Figure 5.2, which clearly demonstrates the execution times for all three versions at different SPP levels.

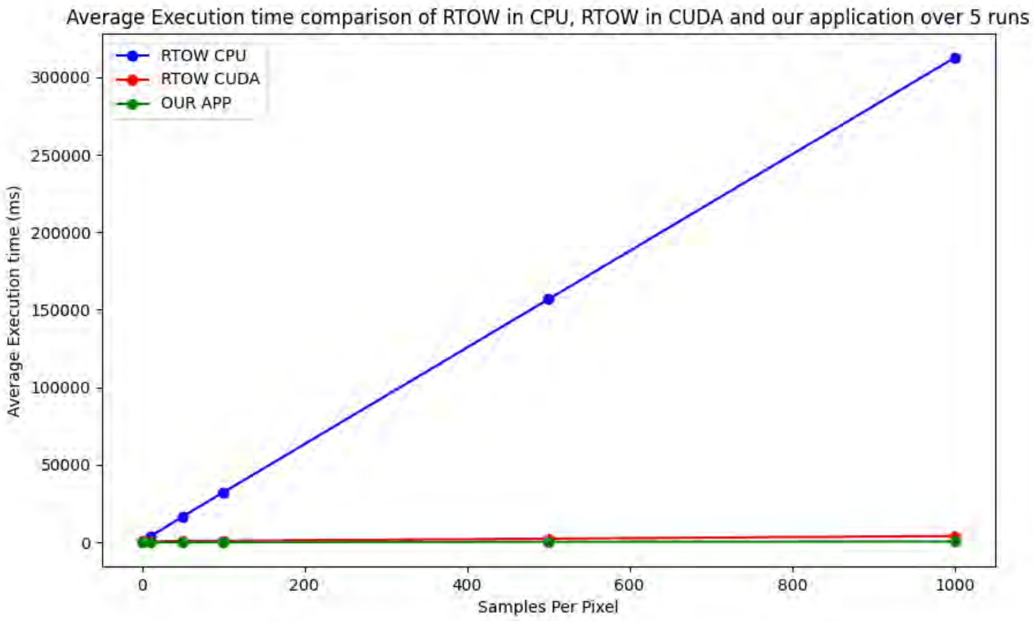


Figure 5.2: Graph between all three implementations

In the graph, the x-axis represents the samples per pixel, while the y-axis represents the average execution time in milliseconds. The blue line denotes the CPU version of RTOW, the red line indicates the CUDA version of RTOW, and the green line represents our application. It is evident from the results that there is a substantial disparity in performance between the CPU and CUDA versions. This distinction, while expected due to the inherent parallel processing capabilities of GPUs, reaffirms the advantageous utilization of CUDA for computationally intensive tasks such as ray tracing.

In addition to the comprehensive comparison among the CPU, CUDA, and our application, we also prepared a more focused comparison specifically between the CUDA version and our application. This approach allows us to observe and analyze the efficiency and improvements brought about by our optimizations more clearly. The graph, as seen in Figure 5.3, generated from this test narrows down the comparison to just the CUDA implementations, thereby eliminating any potential distractions or scaling issues introduced by the vast performance gap between CPU and GPU implementations. This focused comparison provides a clear picture of how our CUDA-based ray tracer stands against the original CUDA version in terms of performance and efficiency. Evidently, the CUDA RTOW execution time is about 8.89 times (or 889%) greater than our application's execution time at 1000 SPP.

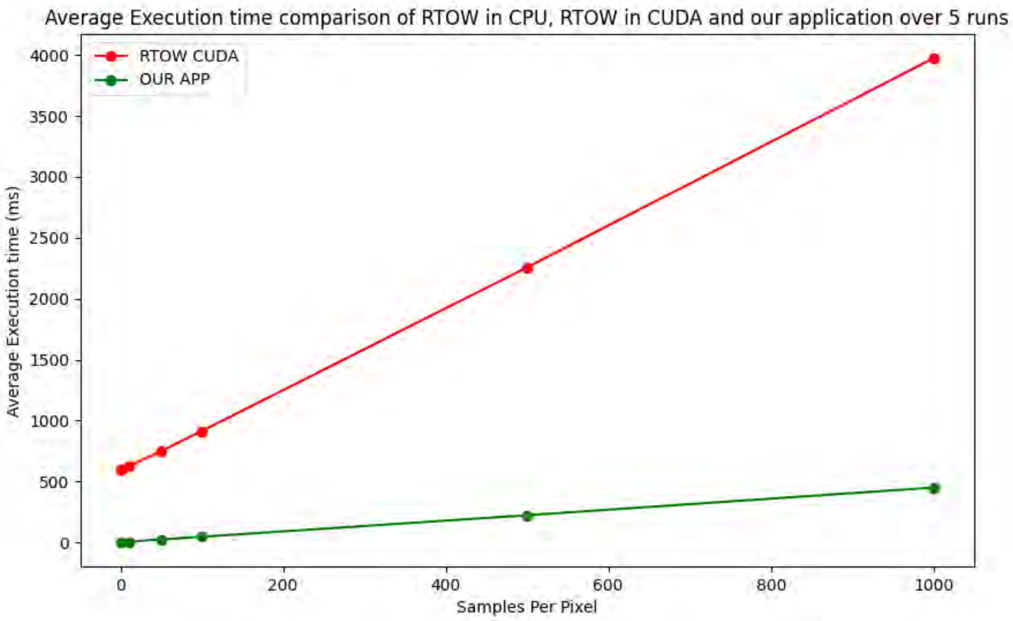


Figure 5.3: Graph between RTOW CUDA and our Application

As the graph illustrates, our CUDA-based ray tracer significantly outperforms the CPU-based version of RTOW across all SPP values. It also showcases competitive performance when compared to the CUDA version of RTOW, demonstrating the efficacy of the various optimizations and design decisions we have made throughout our application's development.

5.2 Conclusions

This thesis has embarked upon a thorough and comprehensive journey into the realm of GPU-based ray tracing, employing CUDA as the framework of choice. Our study began with the fundamentals of ray tracing, its principles, algorithms, and their implications for realistic image rendering. We delved into the intricate physics of light and the mathematics of 3D geometry to understand how ray tracing can simulate the natural behavior of light, creating lifelike digital images that hold true to the physics of the real world.

From there, our investigation transitioned into GPU programming and the CUDA platform. We explored the architecture of modern GPUs, understanding their massive parallelism capabilities, and how this computing power can be harnessed using CUDA. The thesis presented a detailed examination of CUDA's architecture, its memory hierarchy, and the programming model that allows developers to exploit the inherent parallelism in computational problems.

The primary achievement of our work lies in the successful design and implementation of an efficient ray tracer on the CUDA platform. Our ray tracing engine stands as a testament to the potent synergy between the inherent parallelism of ray tracing algorithms and the computational power of modern GPUs. By reimagining a ray tracer in the CUDA environment, we transformed a traditionally resource-intensive, slow rendering process into a real-time graphical engine.

The process of converting our ray tracer to operate efficiently under CUDA provided numerous insights into CUDA optimization techniques. Our project demonstrates how crucial memory access patterns, coalesced memory access, and other CUDA-specific programming strategies are to harness the full power of GPU. Additionally, we successfully showcased how to avoid potential performance pitfalls such as thread divergence and improper use of CUDA's memory hierarchy. The efficient handling of recursion, use of intrinsic functions, avoidance of virtual functions, and the use of the Thrust library are further evidence of the need for a deep understanding of the CUDA environment.

The end result is a robust, efficient, and fast real-time ray tracing application. It serves as a practical example and blueprint for leveraging CUDA's capabilities in creating high-performance graphics applications. Our ray tracer encapsulates the intricate balance between the mathematical rigor of ray tracing, the architectural knowledge of GPU programming, and the practical wisdom of CUDA programming. It stands as an embodiment of the powerful capabilities that CUDA offers to developers venturing into the realm of high-performance computing and real-time graphics rendering.

5.3 Future Work

Our work on developing a CUDA-based ray tracer has been rewarding, and while we have accomplished our primary goal, there are several avenues for improvement and expansion. The current implementation has a few known limitations and potential enhancements that we can explore in future iterations.

5.3.1 Bug Fixes

There are a few identified issues in our current implementation that need to be addressed in future iterations of the application. Fixing these will result in an improved user experience and expanded capability of the ray tracer. Here are some of the main bugs that we have recognized:

- Camera Snapping: Our current implementation sometimes experiences a glitch where the camera abruptly ’snaps’ to a new position or orientation when the user clicks inside the Generated Image window to gain focus. This unintended behavior disrupts the seamless navigation experience and can potentially cause disorientation for the user, especially when they are carefully positioning the camera for a specific view.

- Window Minimization: A persistent issue in our application involves the minimization of the main application window. Currently, when the window is minimized, it does not restore properly, rendering the application inaccessible until it is restarted. This is a significant usability issue that needs to be addressed to improve the overall robustness of the application.

In subsequent updates to our application, we aim to fix these bugs to improve the user interface, increase the application’s stability, and expand its capabilities. This will ensure a smoother and more reliable experience for users while they engage with our ray tracing engine.

5.3.2 Extra Features and Optimizations

There are also several potential features and optimizations that could be added to increase the performance and functionality of the ray tracer.

Firstly, we could introduce more advanced rendering techniques, such as importance sampling [45]. This technique prioritizes the sampling of light that significantly contributes to the final color of a pixel, reducing noise and improving image quality.

Secondly, the addition of more sophisticated ray tracing features like adaptive sampling, sub-surface scattering, or caustics would enhance the visual fidelity of the renderer. Adaptive sampling would improve performance by reducing unnecessary computation, whereas subsurface scattering and caustics would add to the realism by simulating complex light behavior.

At present, our ray tracer builds the BVH tree on the CPU and then transfers it to the GPU for ray-object intersection tests. While this approach serves its purpose, it is not the most efficient way to take advantage of the GPU's computational capabilities. Offloading the BVH construction to the GPU could provide substantial performance benefits, due to the highly parallelizable nature of the process.

Linear Bounding Volume Hierarchies (LBVH) and Spatial Bounding Volume Hierarchies (SBVH) are two strategies that are particularly well-suited for GPU implementation. These techniques build the BVHs in a top-down manner, but unlike traditional methods, they exploit the GPU's capability for parallel processing to create the BVHs more quickly [46]. This, in turn, results in faster render times due to reduced time spent in the construction phase and accelerated intersection tests.

Incorporating LBVH or SBVH techniques into our ray tracer is a complex task, as it would involve significant changes to our existing architecture. However, we believe the performance gains that could potentially be realized are well worth the investment of time and effort.

Adding support for rendering triangles is another significant enhancement we aim to implement in future iterations of our ray tracer. As of now, our application primarily deals with spheres, which limits the complexity of the scenes we can create.

Triangular meshes are a fundamental component of 3D modeling and graphics. The ability to render triangle meshes would not only increase the richness and variety of the scenes our application can handle but would also allow us to import and render complex 3D models from external sources.

Implementing triangle support involves more than just handling a new shape; it requires a robust method for efficiently managing and accessing potentially vast numbers of triangle primitives. Strategies like the use of indexed triangle lists and further enhancements to our

BVH system will likely play a critical role in this feature's successful implementation.

The addition of triangle support is a substantial undertaking that would fundamentally expand our ray tracer's capabilities. However, with the significant potential for improving the visual richness and flexibility of our application, it is a challenge we eagerly anticipate.

Implementing these enhancements will not only improve the visual output and performance of our ray tracer, but will also broaden the application's usability and potential. This will make our ray tracer a more versatile and effective tool for real-time 3D rendering.

5.4 Epilogue

This thesis has been an enlightening journey into the realm of GPU programming and real-time ray tracing. It has provided a thorough understanding of CUDA programming, the intricacies of ray tracing, and how to marry the two for high-performance graphics rendering. The project serves as a foundation for further exploration into advanced GPU-accelerated rendering techniques, paving the way for more realistic, real-time graphics in the future. The experience and knowledge gained through this endeavor are valuable assets for future research and development in the field of real-time rendering and high-performance computing.

Bibliography

[1] Cuda zone. https://developer.nvidia.com/cuda-zone. Accessed: 18-06-2023.

[2] Bounding volume hierarchy. https://en.wikipedia.org/wiki/Bounding_volume_hierarchy. Accessed: 18-06-2023.

[3] Dear imgui. https://github.com/ocornut/imgui. Accessed: 22-04-2023.

[4] Marco Nobile, Paolo Cazzaniga, Daniela Besozzi, Dario Pescini, and Giancarlo Mauri. cutauleaping: A gpu-powered tau-leaping stochastic simulator for massive parallel analyses of biological systems. *PloS one*, 9:e91963, 03 2014.

[5] M. Usman Ashraf, Fathy Eassa, Aiiad Albeshri, and Abdullah Algarni. Performance and power efficient massive parallel computational model for hpc heterogeneous exascale systems. *IEEE Access*, PP:1–1, 04 2018.

[6] Cornell box. https://en.wikipedia.org/wiki/Cornell_box. Accessed: 14-06-2023.

[7] Ray tracing. https://developer.nvidia.com/discover/ray-tracing. Accessed: 06-03-2023.

[8] Cuda toolkit. https://developer.nvidia.com/cuda-toolkit. Accessed: 06-03-2023.

[9] Opengl. https://www.opengl.org/. Accessed: 07-03-2023.

[10] Cuda-opengl interoperability. https://www.3dgep.com/opengl-interoperability-with-cuda/. Accessedς: 07-03-2023.

[11] Cuda c++ programming guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide/. Accessed: 13-03-2023.

[12] An easy introduction to cuda c and c++. https://developer.nvidia.com/blog/easy-introduction-cuda-c-and-c/. Accessed: 20-03-2023.

[13] Cuda samples. https://github.com/NVIDIA/cuda-samples. Accessed: 22-03-2023.

[14] Cuda opengl interop api. https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__OPENGL.html#group__CUDART__OPENGL. Accessed: 22-03-2023.

[15] Peter Shirley. *Ray Tracing in One Weekend*. CreateSpace Independent Publishing Platform, 2016.

[16] Accelerated ray tracing in one weekend in cuda. https://developer.nvidia.com/blog/accelerated-ray-tracing-cuda/. Accessed: 26-03-2023.

[17] Visual studio. https://visualstudio.microsoft.com/. Accessed: 25-04-2023.

[18] Visual studio code. https://code.visualstudio.com/. Accessed: 25-04-2023.

[19] Neovim. https://neovim.io/. Accessed: 25-04-2023.

[20] Nvidia nsight compute. https://developer.nvidia.com/nsight-compute. Accessed: 25-04-2023.

[21] Nvidia nsight graphics. https://developer.nvidia.com/nsight-graphics. Accessed: 25-04-2023.

[22] Amd µprof. https://www.amd.com/en/developer/uprof.html. Accessed: 25-04-2023.

[23] Intel vtune profiler. https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html. Accessed: 25-04-2023.

[24] Glad. https://glad.dav1d.de/. Accessed: 25-04-2023.

[25] An opengl library | glfw. https://www.glfw.org/. Accessed: 25-04-2023.

[26] Opengl mathematics. https://github.com/g-truc/glm. Accessed: 25-04-2023.

[27] Thrust. https://docs.nvidia.com/cuda/thrust/index.html. Accessed: 25-04-2023.

[28] stb - single file public domain libraries for c/c++. https://github.com/nothings/stb. Accessed: 25-04-2023.

[29] spdlog. https://github.com/gabime/spdlog. Accessed: 25-04-2023.

[30] Cmake. https://cmake.org/. Accessed: 25-04-2023.

[31] Premake. https://premake.github.io/. Accessed: 25-04-2023.

[32] F. Gonzalez and G. Patow. Continuity and interpolation techniques for computer graphics. *Comput. Graph. Forum*, 35(1):309–322, feb 2016.

[33] Opengl shading language. https://www.khronos.org/opengl/wiki/Core_Language_(GLSL). Accessed: 24-04-2023.

[34] Imgui image loading and displaying examples. https://github.com/ocornut/imgui/wiki/Image-Loading-and-Displaying-Examples. Accessed: 22-04-2023.

[35] Lambertian reflectance. https://en.wikipedia.org/wiki/Lambertian_reflectance. Accessed: 22-04-2023.

[36] Specular reflection. https://en.wikipedia.org/wiki/Specular_reflection. Accessed: 22-04-2023.

[37] Fresnel equations. https://en.wikipedia.org/wiki/Fresnel_equations. Accessed: 25-04-2023.

[38] Schlick's approximation. https://en.wikipedia.org/wiki/Schlick%27s_approximation. Accessed: 25-04-2023.

[39] Peter Shirley. *Ray Tracing The Next Week.* CreateSpace Independent Publishing Platform, 2016.

[40] Jonathan Passerat-Palmbach, Jonathan Caux, Pridi Siregar, Claude Mazel, and David Hill. Warp-level parallelism: Enabling multiple replications in parallel on gpu. *ESM 2011 - 2011 European Simulation and Modelling Conference: Modelling and Simulation 2011*, 01 2015.

[41] How to access global memory efficiently in cuda c/c++ kernels. `https://developer.nvidia.com/blog/how-access-global-memory-efficiently-cuda-c-kernels/`. Accessed: 23-05-2023.

[42] Tail call. `https://en.wikipedia.org/wiki/Tail_call`. Accessed: 26-05-2023.

[43] Alejandro Segovia, Xiaoming Li, and Guang Gao. Iterative layer-based raytracing on cuda. In *2009 IEEE 28th International Performance Computing and Communications Conference*, pages 248 – 255, 01 2009.

[44] Mengchi Zhang, Ahmad Alawneh, and Timothy G. Rogers. Judging a type by its pointer: Optimizing gpu virtual functions. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS ’21, page 241–254, New York, NY, USA, 2021. Association for Computing Machinery.

[45] Christoph Peters. Brdf importance sampling for polygonal lights. *ACM Trans. Graph.*, 40(4), jul 2021.

[46] Thinking parallel, part iii: Tree construction on the gpu. `https://developer.nvidia.com/blog/thinking-parallel-part-iii-tree-construction-gpu/`. Accessed: 10-06-2023.

[47] Github repository. `https://github.com/Trippasch/CudaRayTracer`. Accessed: 14-06-2023.

[48] Gnu general public license. `https://www.gnu.org/licenses/gpl-3.0.txt`. Accessed: 14-06-2023.

APPENDICES

Appendix A

Software Documentation

A.1 Installation

A.1.1 System Requirements

- NVIDIA CUDA-capable GPU
- CUDA SDK

A.1.2 Installation Steps

All the installation steps of the ray tracing application can be found on the online GitHub repository [47].

A.2 Usage

Once installed, you can run the application via the command line, using the executable created during the build process. More information can be found on the GitHub project page [47].

A.3 Contact Information

For further information or if you have any questions, inquiries, or feedback related to the project, please feel free to contact the project maintainers:

Paschalis Choropanitis

- Email: pchoropan@uth.gr
- GitHub: `https://github.com/Trippasch`

Panayiotis Yiannoukkos

- Email: pgiannoukkos@uth.gr
- GitHub: `https://github.com/pgiannoukkos`

Please respect that this is a voluntary project, and response times may vary. For bug reports or feature requests, consider opening an issue directly on the GitHub project page [47].

A.4 License

The software is released under the GNU General Public License version 3 (GPLv3). This is a strong copyleft license that requires anyone who distributes the code or a derivative work to make the source available under the same terms. It also provides an express grant of patent rights from contributors to users.

The full license text is included in the LICENSE file in the root directory of the project's source code. The terms of the GPL ensure that the software remains free and open-source, enabling users and contributors to use, modify, and distribute the software while keeping it open and free for all.

For the full details of the GNU General Public License version 3 (GPLv3), you may refer to the official GNU website [48] or the LICENSE file in our repository. For any legal advice, please consult with a legal expert.

Appendix B

Images

B.1 Sample Test Scenes

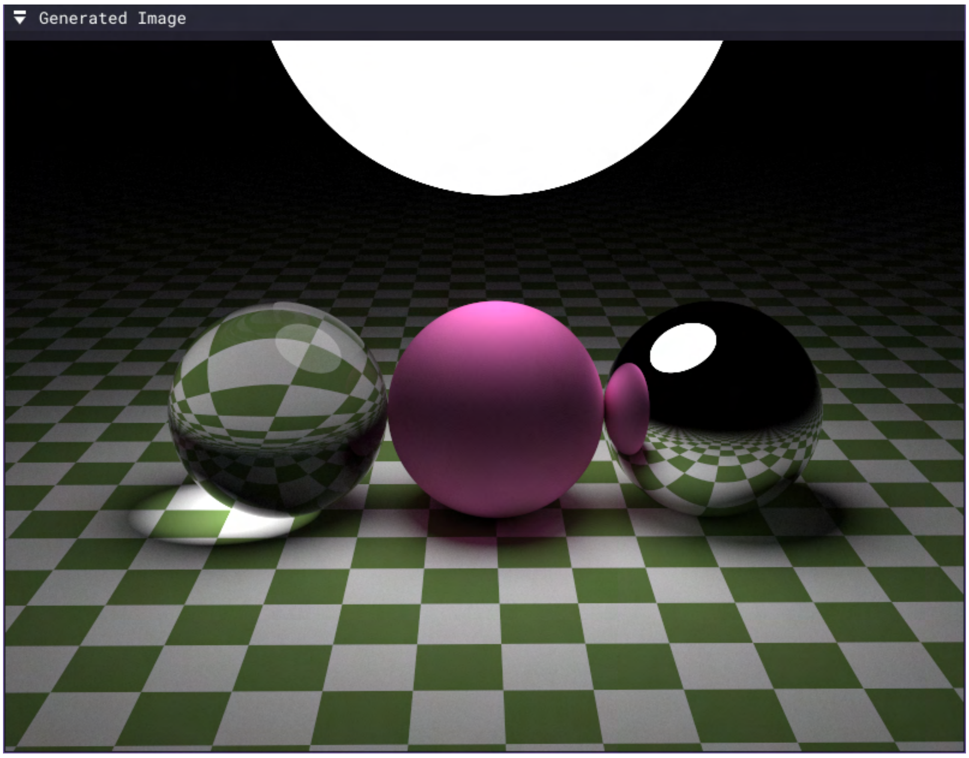


Figure B.1: Scene featuring four spheres with all the supported materials

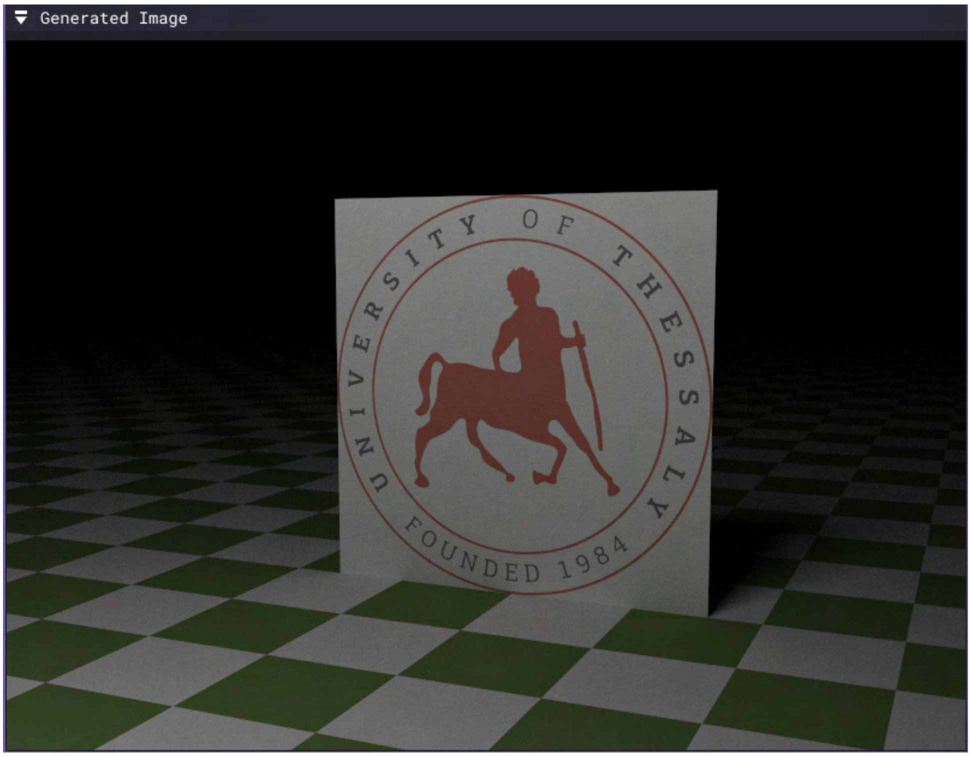


Figure B.2: Scene featuring a textured rectangle with the logo of the University of Thessaly

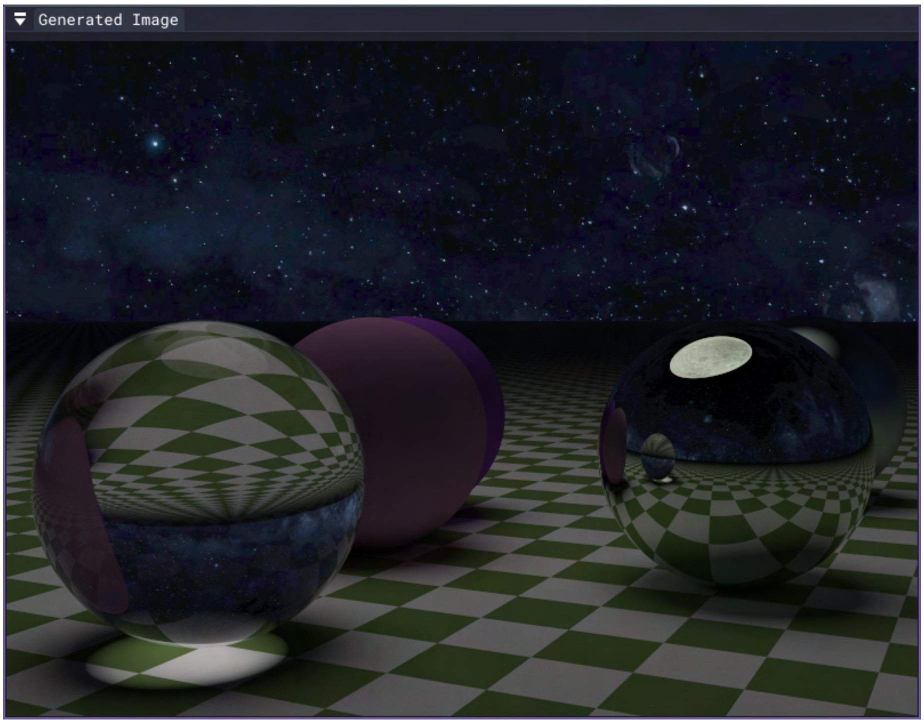


Figure B.3: Scene featuring the Milky Way as a skybox

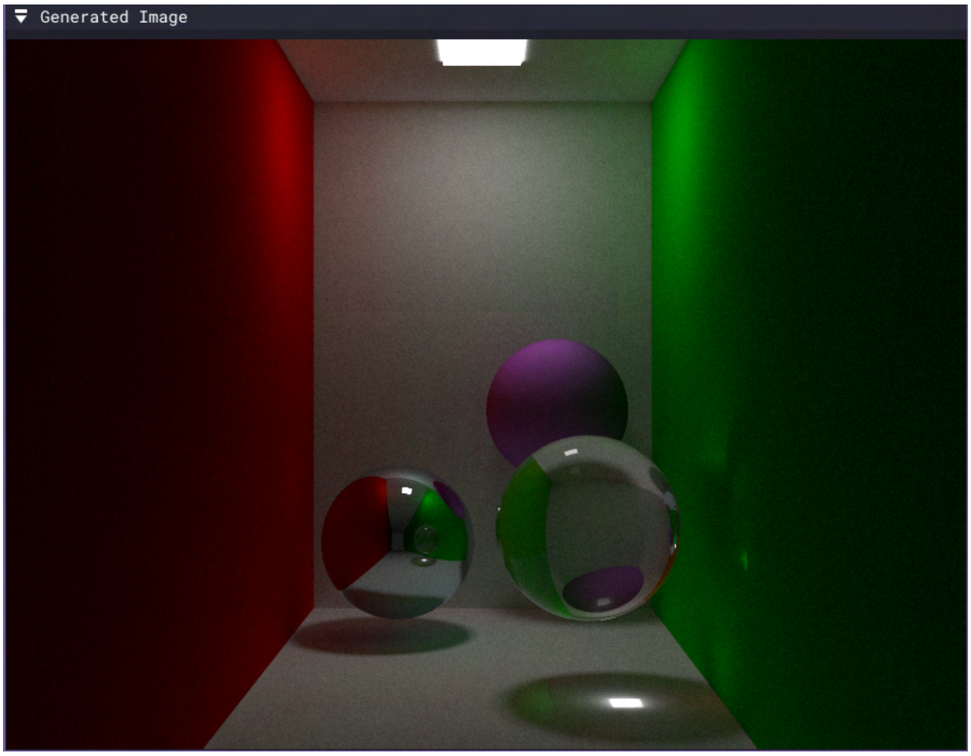


Figure B.4: Scene featuring a Cornell Box [6] with three spheres and a light inside it

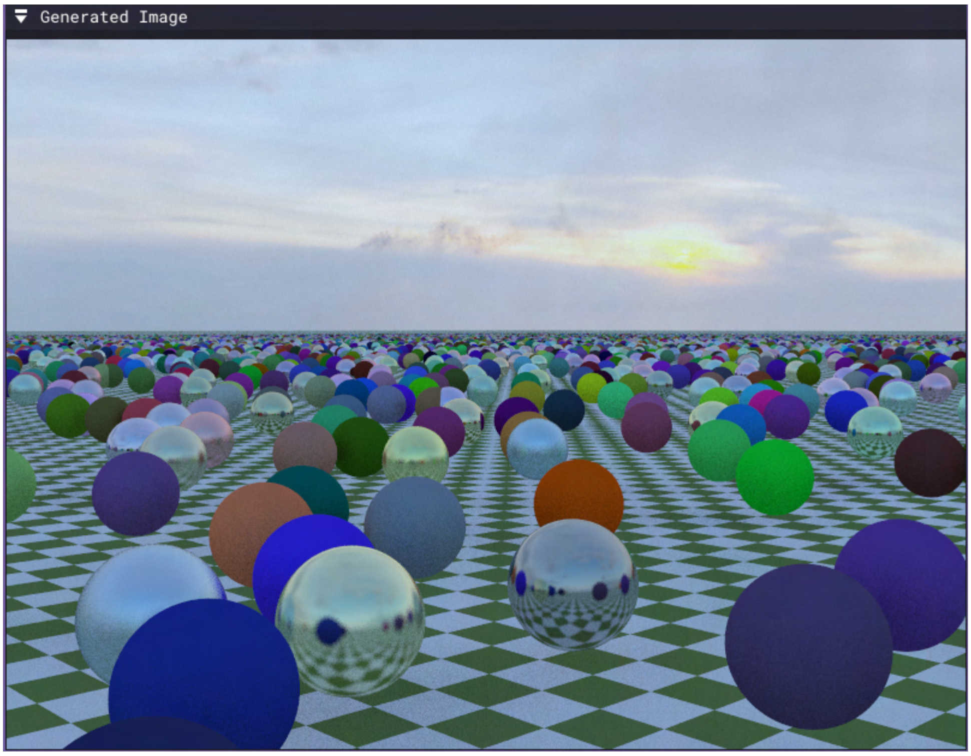


Figure B.5: Scene featuring 10000 spheres with a sunset sky as a skybox