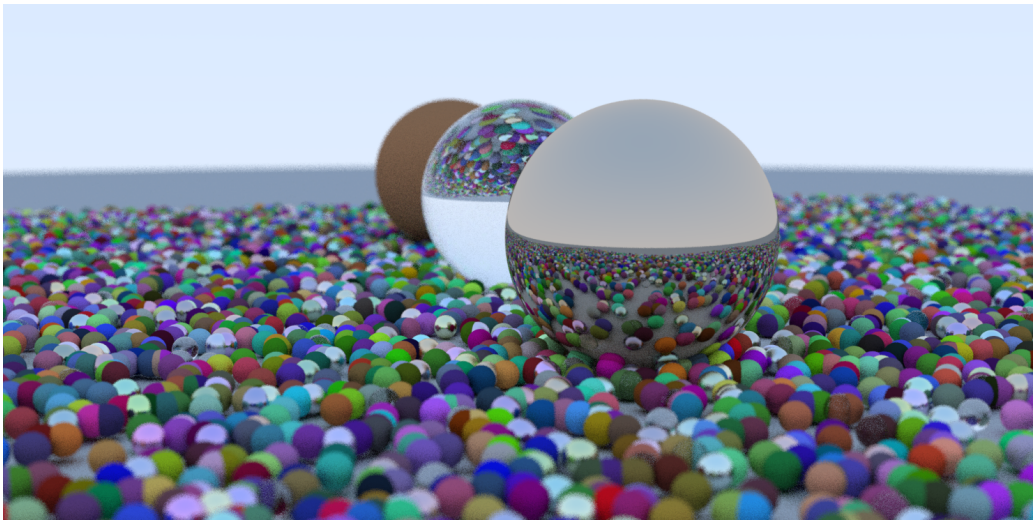
Optimizing a CUDA Path Tracer
Final Project Report

DD2360 Applied GPU Programming – HT24 – KTH

Christiane Kobalt kobalt@kth.se
Nicolas Müller npgmu@kth.se
Maximilian Ranzinger maxran@kth.se
Robin Sieber rjsieber@kth.se



Abstract

In this report we discuss the optimization of an existing NVIDIA ray tracing application. We use profiling to identify performance bottlenecks and employ different optimization strategies to address them.

In a first optimization attempt we utilize reduced floating point precision to boost performance, which held rather disappointing results.

The second optimization strategy, and the core of our work, consists of an acceleration structure, namely an octree, implemented on top of the preexisting code. The use of such a data structure allows us to improve the computational complexity of the rendering process and results in a significant speedup compared to the base application.

In the following we cover the theory behind general ray tracing algorithms and acceleration structures. We then present the methodology and software implementation behind our optimization, as well as the experimental setup used for testing and benchmarking. Finally, we show our results and discuss improvements, limitations and possible further improvements.

Contributions

Christiane Kobalt: Progressive refinement and live preview window using OpenGL; CMake build environment; design of octree data structure and construction of the tree on the CPU side; corresponding chapters in the report, introduction chapter.

Nicolas Müller Ensuring Colab compatibility; initial profiling; output modes; debugging and optimization of octree implementation; world initialization; corresponding chapters in the report; discussion and conclusion.

Maximilian Ranzinger: Testing and benchmarking; visualization and interpretation of measurements; experimental setup chapter; results chapter.

Robin Sieber: Initial profiling; implementation of half precision floating point operations; completion and optimization of octree implementation on GPU side; corresponding chapters in the report.

1 Introduction

1.1 From the Rendering Equation to Path Tracing

Modern photorealistic rendering techniques aim to simulate the behavior of light as accurately as possible. One of the most general formulations of the light-transport problem is given by the rendering equation, introduced by Kajiya in 1986 [Kaj86]. In essence, the rendering equation expresses the radiance L leaving a point in a given direction as a combination of emitted light and reflected light that originates from all other surfaces in the scene:

$$L(x, \omega_o) = L_e(x, \omega_o) + \int_\Omega f_r(x, \omega_i, \omega_o)\, L(x', -\omega_i)\, G(x, x')\, d\omega_i \tag{1}$$

$L(x, \omega_o)$ is the outgoing radiance at point x in direction ω_o.

$L_e(x, \omega_o)$ is the emitted radiance (e.g., from a light source).

$f_r(x, \omega_i, \omega_o)$ is the Bidirectional Reflectance Distribution Function (BRDF), describing how light scatters at x.

$G(x, x')$ is a geometric term accounting for visibility and geometry factors between x and the other points x'.

Ω is the hemisphere (or sphere, depending on notation) of incoming directions ω_i.

Although this integral equation accurately captures the phenomenon of global illumination, solving it analytically for complex 3D scenes is intractable. Path tracing tackles this problem by simulating light transport through random sampling of light paths in the scene. Path tracing was popularized by Kajiya as well, in the same paper [Kaj86], and remains one of the most robust methods for physically based rendering.

1.2 Monte Carlo Integration and Progressive Refinement

At the heart of path tracing lies Monte Carlo integration, which approximates the integral of the rendering equation by shooting many random rays (or "path") and statistically estimating the average amount of radiance observed from each pixel. A simple Monte Carlo estimator of an integral I looks like $I \approx \frac{1}{N} \sum_{i=1}^{N} f(x_i)$ where $f(x_i)$ represents the integrand (in rendering, it would be the contribution of the light from a particular ray or path), and x_i are random samples drawn from a chosen distribution.

In path tracing, each ray trace from camera to scene (and bouncing around until it either leaves the scene or hits a light source) provides one sample. The result for each pixel converges to the true solution given enough samples. However, any Monte Carlo approach inherently introduces noise in the initial render, which diminishes over time as more samples are accumulated. This is called progressive refinement. The advantage is that programmers or artists can see a rough preview of the final render quickly and let the computation continue until acceptable quality is reached.

1.3 The Importance of GPUs for Path Tracing

While path tracing can produce stunningly photorealistic images, the computational cost is high. Generating millions to billions of rays per second for complex scenes can challenge even modern high-performance CPUs. This is where GPUs play a pivotal role. GPUs are originally designed for fast rasterization but due to their massively parallel architecture are now also extensively used for general-purpose parallel tasks such as ray tracing. Their wide vector units and thousands of parallel threads can accelerate path tracing significantly.

A variety of graphics APIs support GPU-based ray tracing and path tracing:

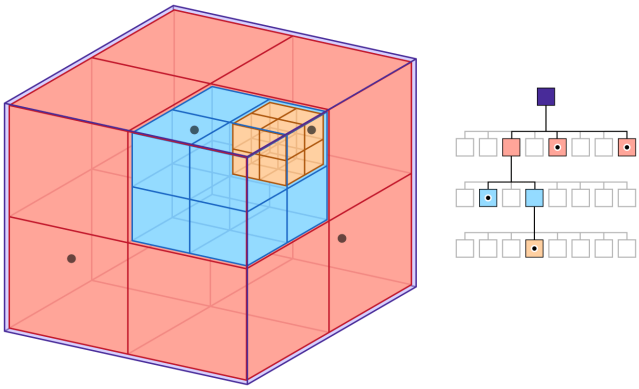


Figure 1: A visual representation of an Octree. [App]

- **DirectX Raytracing (DXR):** Microsoft's extension to DirectX that enables hardware-accelerated ray tracing [Tea18].
- **Vulkan Ray Tracing:** Khronos Group's official extension to the Vulkan API for ray tracing [Gro25].
- **OptiX:** NVIDIA's ray-tracing engine built on CUDA for high-performance ray tracing on NVIDIA GPUs [NVI25].

One of the most flexible and low-level approaches is using CUDA, which exposes the GPU's raw parallelism to the programmer. Developers can implement custom path tracers by directly managing threads and memory on NVIDIA GPUs.

1.4 Acceleration Structure: Octree

When computing millions of ray-scene intersections, a naïve approach that tests every ray against every object is prohibitively expensive with $O(N \times R)$ complexity, where N is the number of objects and R is the number of rays).

Acceleration structures are data structures designed to speed up intersection queries by reducing the number of intersection tests. Two widely used acceleration structures are Bounding Volume Hierarchies (BVHs) and k-d trees. These hierarchical data structures partition 3D space or group geometry in such a way that large subsets of the scene can be quickly tested for intersection or skipped entirely.

An octree is a tree-based data structure that recursively subdivides 3D space into eight smaller cubic regions at each level (see fig. 1). Construction proceeds by splitting any node containing more than a threshold number of objects into eight children. This process continues until each leaf node holds a sufficiently small number of objects or a maximum depth is reached.

When performing intersection tests, the algorithm checks which of the eight child regions the ray intersects and only recurses into those. By pruning large empty regions, octrees reduce the total number of intersection checks, improving the performance of ray tracing—particularly in scenes where geometry is relatively evenly distributed.

1.5 Our Code Base

We chose to rely on the existing code base by NVIDIA developer Roger Allen, whose article was linked on Canvas [All18]. This blog post is again based on the book *Raytracing in One Weekend* by Shirley [Shi20]. The base implementation used for this project is the branch `ch12_where_next_cuda` of Allen's GitHub repository [All], which provides a fully functional CUDA ray tracing implementation. The program renders a scene with multiple spheres of different material and sizes. It parallelizes the rendering process by launching one thread for each pixel in blocks of 8×8 threads. We did not change this configuration and aimed solely at improving the `render` kernel itself.

The source code for this project containing all our optimizations can be found on GitHub as well [Kob+]. The repository includes a `Readme.md` with usage instructions and a Jupyter notebook to effortlessly run everything

inside Google Colab. The code is carefully documented wherever it was adapted for this project (we did not further document the base implementation).

2 Methodology

2.1 Progressive Refinement for Real-Time Preview

To enable an interactive preview that improves over time, we introduced progressive refinement by adding a new kernel, `render_progressive`, which handles exactly one sample per pixel each invocation. After each pass, the partial results stored in `fb` (the framebuffer) are uploaded into an OpenGL texture, and a simple quad is rendered in a GLFW-created window. This allows the scene to gradually converge as more passes accumulate.

In practice, OpenGL window creation and texture updates happen inside the `render_in_window` function, guarded by `#ifdef USE_OPENGL`. This conditional compilation is important because environments such as Google Colab (which often do not support a full GPU driver stack or an active display) cannot compile or run OpenGL code. Hence, the windowed preview is only built when running on a local GPU machine to ensure code portability across different platforms.

2.2 Profiling

We used Nsight Compute and Nsight Systems to profile the base implementation. The average runtime for the different kernels is shown in Figure 2. The `render` kernel clearly dominates the execution time.

The `render` kernel, the only kernel actually running concurrently, is already well optimized regarding occupancy (71.82% achieved vs 87.50% theoretical occupancy). All other kernels are only launched with a single thread for initialization and finalization purposes. Possible benefits from improved scheduling and workload balance are thus very limited. This means that optimization needs to happen on the algorithmic level instead.

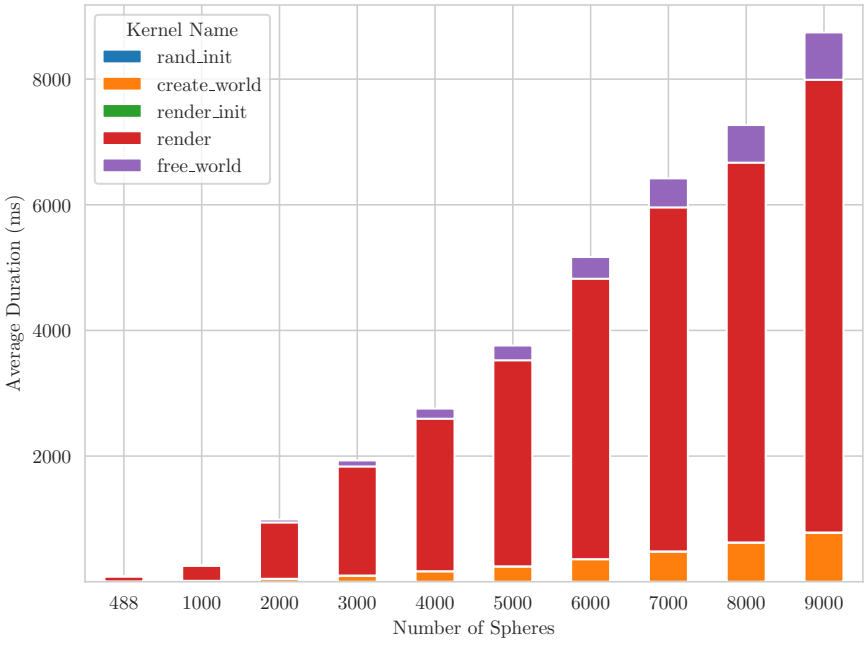


Figure 2: The stacked bar chart displays the distribution of runtime across the kernels of the application for different sphere counts. The sphere radius is set to 0.1.

2.3 Accelerating Floating Point Operations by Reducing Precision

Given the results of the initial profiling and the lack of obvious memory transfer and access optimizations, we set the goal to accelerate the floating point operations themselves. Since half precision floating point numbers only take up 16 bits, we expected a speedup due to faster memory transfer (less bandwidth pressure), better cache utilization (more numbers fit in the same cache size), as well as twice the throughput when handling FMA operations[1].

For the technical implementation, we relied on NVIDIA's `cuda_fp16.h` implementation, which provides intrinsics[2] around which we built a wrapper class `real_t` containing overloaded operators and constructors for ease of use and to keep the necessary code changes minimal. We then replaced every floating point number with the `real_t` datatype. This allows us also to disable the class using preprocessor macros (`USE_FP16`) and use `typedef float real_t;` instead to easily change back to more precise computations. Other minor refactoring tasks included replacing constexpr with const since nvcc does not provide support for half precision constexpr types.

For most functions we need to differentiate whether they are run on the GPU or CPU since the half precision intrinsics are not supported on the latter. Unfortunately, it is not possible to overload those functions using separate `__host__` and `__device__` specifiers. Instead, this can be done using the `__CUDA_ARCH__` macro:

```
__host__ __device__ __forceinline__ real_t operator+(const real_t &other) const {
#ifdef __CUDA_ARCH__
        return real_t(__hadd(val, other.val));
#else
        return real_t(__half2float(val) + __half2float(other.val));
#endif
}
```

In some other cases, more differentiation is necessary when certain functions are not available for half precision, such as $\tan$ in this example:

```
#ifdef __CUDA_ARCH__
    #ifdef USE_FP16
        real_t half_height = real_t(hsin(arg.val) / hcos(arg.val));
    #else
        real_t half_height = tan(arg);
    #endif
#else
    real_t half_height = tan(arg);
#endif
```

2.4 Acceleration Structure: Octree Implementation

In order to implement the algorithmic accelerations described in section 1.4, we mainly target the function `color`, which determines the color for each pixel by following the ray's path for 50 bounces. The naïve implementation checks for each of the 50 reflections if they intersect with any of the spheres in the world by always iterating through the complete list of hitable objects. In our optimized implementation, however, we let the ray traverse through the octree instead, determining which leaf-nodes of the tree are intersected by the ray. This then allows us to efficiently gather the spheres stored in those leaves that possibly allow for a hit and discard all other spheres. In expectation, this should drastically lower the time spent on intersection tests, though for the price of the tree traversal.

We developed a lean C++ Octree class with configurable depth and leafs using macros. The tree is constructed and filled in with the spheres once on the CPU as an additional preprocessing step, before being copied to the GPU, where it can then be used in the render kernel. For the tree to be easily transferable to GPU with a single call to `cudaMemcpy`, we have to avoid typical pointer-based implementations (due to different memory spaces between host and device) and revert to a design relying only on arrays and indices.

[1] `https://developer.nvidia.com/blog/mixed-precision-programming-cuda-8/`
[2] `https://docs.nvidia.com/cuda/archive/11.7.1/cuda-math-api/group__CUDA__MATH__INTRINSIC__HALF.html`

The Octree itself stores the two global arrays with all (internal) nodes and leaves of the tree. The nodes themselves then keep track of their children via their indices in those arrays. Leaves are treated differently than internal nodes because they have to store the spheres they contain (again via their indices in the global sphere list).

```
struct Octree {
    OctNode nodes[NUMBER_NODES];           ///< Array of all nodes in the tree
    OctLeaf leaves[NUMBER_LEAFS + 1];      ///< Array of all leaves
    int nodeCount = 0;                     ///< Number of nodes currently in use
    int leafCount = 1;                     ///< Number of leaves currently in use
};
struct OctNode {
    int level;                             ///< Current depth in the tree
    AABB aabb;                             ///< Bounding box for this node
    int children[CHILDREN_COUNT];          ///< Indices of child nodes or leaves
};
struct OctLeaf {
    int sphere_indices[SPHERES_PER_LEAF];  ///< Indices of spheres in this leaf
    int index_count;                       ///< Number of spheres currently stored
};
```

3 Experimental Setup

We developed and tested the code on Google Colab (T4 GPU) as well as on local GPUs, namely an RTX 4080 and RTX 3080. The final experiments in this report were conducted using an RTX 4080. We used CMake to ensure portability between platforms. No third-party libraries except simple CUDA headers for randomness, timing and FP16 were used.

The code takes no input. A 1200×800 pixel image is generated, featuring three medium-sized spheres and multiple smaller spheres that display reflections created through ray tracing. The spheres are placed randomly according to a uniform distribution on a square surface or world. The number of spheres and their radius are hardcoded, any changes to those parameters require a recompilation of the code.

For profiling, NVIDIA Nsight Systems (`nsys`) and NVIDIA Nsight Compute CLI (`ncu`) were used. Each profiling attempt or experiment was repeated five times to obtain consistent measurements of the metrics. The primary metric of interest is the runtime of the `render` kernel. Differences in picture quality is assessed through the structural similarity index measure (SSIM). The SSIM[3] compares two images by evaluating their luminance, contrast, and structural patterns to reflect perceptual differences. SSIM values range from -1 to 1, where 1 indicates identical images, 0 no similarity and -1 means the images are perfectly inverted.

The base implementation by [All] (see section 1.5), henceforth also referred to as the baseline, serves as the reference for comparing the optimization experiment involving half-precision. Due to the introduction of the octree, as well as the ability to vary sphere quantities and sizes, the baseline required slight modifications. Therefore, the results between FP16 and the octree are not directly comparable.

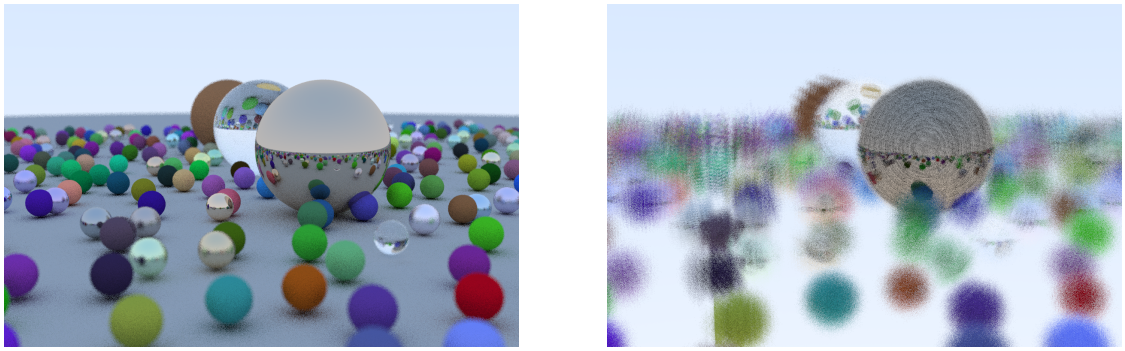
4 Results

Our first attempt to improve performance was to change calculations from single to half precision, as outlined in section 2.3. By doing that we achieved a speedup of $1.329\times$ from 322.12 ms down to 242.39 ms. A Welch's t-test showed a significant reduction in runtime ($p = 7.51 \cdot 10^{-7}$), confirming the performance improvement is statistically significant and unlikely due to random variation. Furthermore, the memory throughput increased from 53% to 90%, as expected in section 2.3. However, the image quality decreased drastically with an SSIM value of only 0.432. Figure 3 shows calculating rays in half precision results in a distorted and noisy image.

The drop in quality is due to the limited precision of FP16 calculations, which introduce small errors in ray-surface intersections and light transport. These errors amplify with each ray bounce, causing artifacts like

[3] `https://scikit-image.org/docs/0.24.x/auto_examples/transform/plot_ssim.html`

distorted reflections, light leakage, and missing shadows. The lack of precision makes the image look grainy (see section 1.2). Consequently, while the speedup is beneficial, the trade-off in visual fidelity is significant, as demonstrated by the low SSIM value and the distorted appearance of the FP16-rendered image. Therefore, we discontinued optimization using half-precision and instead implemented an octree to reduce runtime.



(a) Single Precision (Baseline)

(b) Half Precision

Figure 3: Comparison of the output image of the application with single and half precision

Figure 4 illustrates the progression of the average runtime for the `render` kernel over five runs, conducted across multiple experiments with varying numbers of spheres. Initially, for lower sphere counts (488 and 1000 spheres), the overhead introduced by utilizing the octree structure leads to an increase in the `render` kernel runtime. This indicates that for smaller number of spheres, the cost of managing the octree outweighs its benefits. However, as the number of spheres grows, the octree begins to demonstrate significant performance advantages. The speedup factor increases consistently with the number of spheres, ultimately reaching a maximum speedup of 5.4 when rendering 8000 spheres. Welch t-test's comparing the baselines and with the octree implementation showed a significant reduction in runtime, confirming the performance improvement is statistically significant and unlikely due to random variation.

For spheres of larger radius, the improvement achieved with the octree is less significant compared to that observed with smaller spheres. This is due to the influence of octree hyperparameters, such as tree depth and the number of spheres per leaf, which require careful tuning for the specific use case. These parameters govern the number of spheres the application can manage in terms of memory. A smaller number of leaves reduces duplication, while a larger number of leaves increases it. As the number of leaves increases, the area each leaf covers decreases, leading to greater duplication when larger spheres intersect multiple leaves. This duplication negatively impacts performance. In our configuration, the octree performs more effectively with smaller spheres and is limited to managing approximately 9000 spheres.

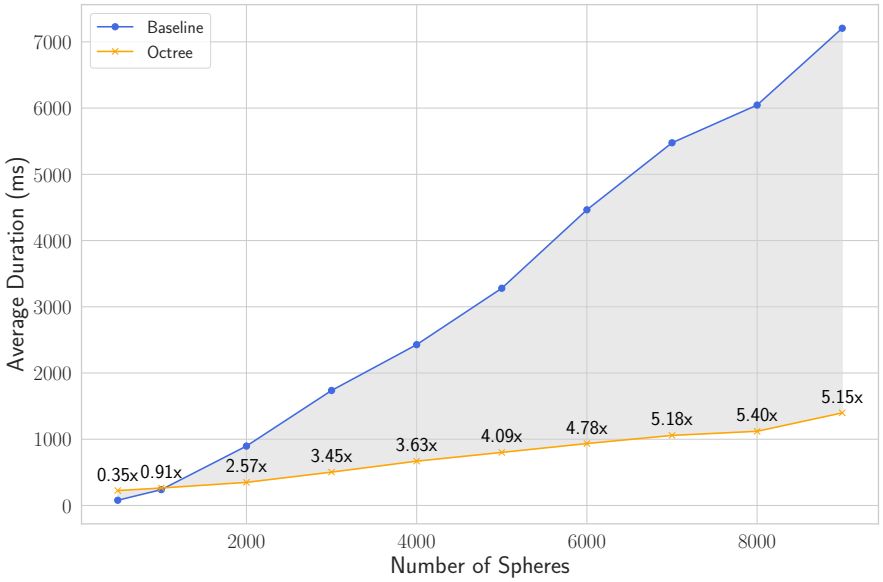


Figure 4: The line chart illustrates the evolution of the `render` kernel runtime for the application, comparing the octree approach to the baseline. The values presented indicate the speed-up achieved by the octree. Each data point is the statistical average of $N = 5$ measurement runs. The sphere radius is set to 0.1.

5 Discussion and Conclusion

In this project we explored two different approaches to optimize a CUDA-based path tracer: reduced floating-point precision and acceleration using an octree data structure.

Because the base implementation was already very well designed, there were no obvious optimizations regarding memory transfer or workload balance. Instead, we had to aim for a speedup on the algorithmic level.

While the FP16 implementation achieved a $1.329\times$ speedup, the significant reduction in image quality (SSIM of 0.432) made this approach impractical for any real-world application. The visual artifacts and increased noise demonstrated that path tracing algorithms require the precision offered by 32-bit floating point operations to maintain acceptable image quality.

The octree implementation proved more successful, achieving up to $5.4\times$ speedup for scenes with a large number of spheres. For scenes with fewer than ~ 1000 spheres, the overhead of tree traversal still outweighed the benefits of reduced intersection tests.

Possible opportunities for further improvements include:

- Redesigning the octree implementation to be more modular, including adaptive tree depth based on local object density
- Optimizing memory usage of the octree
- Parallelizing the octree construction process
- Investigating possible unified memory based implementations

Taking on the task of optimizing a CUDA code written by a developer who works at NVIDIA himself certainly proved to be challenging. Nonetheless, we were able to achieve great results and are happy about the outcome of this project.

Expected Grade: A

References

Allen, Roger. *Accelerated Ray Tracing in One Weekend in CUDA*. Nov. 2018. URL: https://developer.nvidia.com/blog/accelerated-ray-tracing-cuda/.

— *GitHub repository: Base implementation*. URL: https://github.com/rogerallen/raytracinginoneweekendincuda/tree/ch12_where_next_cuda.

Apple. *Octree Visualization*. https://docs-assets.developer.apple.com/published/183088f967/octree_2x_d5ec086e-6563-4f2b-99a2-4e1762919c72.png. Accessed: 2025-01-01.

Group, Khronos. *Vulkan Specification*. https://docs.vulkan.org/spec/latest/chapters/raytracing.html. Accessed: 01.01.25. 2025.

Kajiya, James T. "The rendering equation". In: *Proceedings of the 13th annual conference on Computer graphics and interactive techniques* (1986).

Kobalt, Christiane et al. *Project GitHub repository*. URL: https://github.com/MuellerNico/DD2360-RayTracing.

NVIDIA. *NVIDIA OptiX™ Ray Tracing Engine*. https://developer.nvidia.com/rtx/ray-tracing/optix. Accessed: 01.01.25. 2025.

Shirley, Peter. *Ray Tracing in One Weekend*. 4.0.1. 2020. URL: raytracing.github.io/books/RayTracingInOneWeekend.html.

Team, D3D. *Announcing Microsoft DirectX Raytracing!* https://devblogs.microsoft.com/directx/announcing-microsoft-directx-raytracing/. Accessed: 01.01.25. 2018.