BERGISCHE UNIVERSITÄT WUPPERTAL

Fakultät für Mathematik und Naturwissenschaften

Lehrstuhl für Angewandte Informatik

PhD Thesis

Advancing Ray-Traced Ultra-High Vacuum Simulations: Enhanced Algorithms and Data Structures in Molflow

Pascal Bähr
Informatik
Computer Science

Wuppertal, den 06. Dezember 2023

Abstract

Vacuum components play a critical role in many fields. Their design can be a tedious progress, as they are usually evaluated throughout many iterations with simulation software such as Molflow. Molflow uses the Test Particle Monte Carlo (TPMC) method to approximate physical quantities such as the pressure or particle density in arbitrary complex geometries. The TPMC traces independent particles inside a geometry using a ray tracing based algorithm to gather the corresponding statistics. The performance of these Monte Carlo simulations largely depends on the efficiency of the underlying data structure and algorithms used for the ray tracing query. As ray tracing kernels reached new heights leading to real-time rendering with recent advancements in both algorithm research and specialised GPU hardware, the utility of these state-of-the-art methods has been investigated and is evaluated for their suitablity for physical simulations such as Molflow. We provide enhanced and newly developed algorithms and data structure for Molflow's Monte Carlo model, in particular the time-dependent simulations. A specialised ray tracing kernel is developed based on our findings for both CPU-driven simulations leading to an application suitable for HPC environments, as well as GPU simulations where the design of a CUDA kernel is backed by NVIDIA's OptiX API for ray tracing to leverage hardware-acceleration utilising ray tracing units (RTUs) found on modern NVIDIA RTX GPUs. Further, we developed two splitting criteria to construct performant acceleration data structure, the adapted Ray Distribution Heuristic and the Hit Rate Heuristic. Both methods are designed to leverage Molflow's statistical nature. The developed solution for the GPU utilizes a newly developed offset to mitigate negative effects that arise from 32-bit floating point operations that are inherently used for hardware-accelerated ray tracing. Our GPU kernel utilizes RTUs as much as possible. The Neighbor Aware Offset handles some of the effects on the software side. Our research represents the initial step towards enabling Molflow simulations on GPUs.

Kurzfassung

Vakuumkomponenten spielen in vielen Bereichen eine entscheidende Rolle. Ihr Entwurf kann ein mühsamer Prozess sein, da sie üblicherweise durch viele Iterationen mit Simulationssoftware wie Molflow entwickelt werden. Molflow verwendet die Test-Partikel-Monte-Carlo (TPMC) Methode, um physikalische Größen wie Druck oder Partikeldichte in beliebig komplexen Geometrien zu approximieren. Die TPMC verfolgt unabhängige Partikel innerhalb einer Geometrie unter Verwendung eines auf Ray-Tracing basierenden Algorithmus, um die entsprechenden Statistiken zu sammeln. Die Performance dieser Monte-Carlo-Simulationen hängt in großem Maße von der Effizienz der zugrundeliegenden Datenstrukturen und Algorithmen ab, die für die Ray-Tracing Routinen verwendet werden. Dedizierte Ray-Tracing Einheiten lassen das Echtzeitrendering wieder relevanter werden. Die jüngsten Fortschritte sowohl in der Algorithmenforschung als auch in spezialisierter GPU-Hardware haben uns dazu veranlasst, den Nutzen dieser Methoden zu untersuchen und deren Eignung für physikalische Simulationen wie Molflow zu bewerten. Wir haben verbesserte und neu entwickelte Algorithmen und Datenstrukturen für Molflows Monte-Carlo-Modell implementiert, insbesondere für die Simulationen mit Zeitparameter. Spezialisierte Ray-Tracing Algorithmen wurden basierend auf diesen Erkenntnissen sowohl für CPU-gesteuerte Simulationen entwickelt, was die Anwendung in HPC-Umgebungen ermöglicht, als auch für GPU-Simulationen, bei denen das Design eines CUDA-Programms durch NVIDIAs OptiX-API für Ray-Tracing unterstützt wird, um hardwarebeschleunigte Ray-Tracing-Einheiten (RTUs) auf modernen NVIDIA RTX GPUs zu nutzen. Darüber hinaus haben wir zwei Splitting-Kritierien entwickelt, um leistungsfähige Beschleunigungsdatenstrukturen zu konstruieren, die angepasste Ray Distribution Heuristic (RDH) und die Hit Rate Heuristic (HRH). Beide Methoden wurden entwickelt, um das statistische Fundament von Molflow zu nutzen. Die für die GPU entwickelte Lösung verwendet einen neu entwickelten Offset, um negative Effekte von 32-Bit-Gleitkommaoperationen zu reduzieren, die mit hardwarebeschleunigtem Ray-Tracing einhergehen. Unser GPU-Code maximiert die Nutzung von RTUs. Der Neighbor Aware Offset behandelt einige dieser Effekte auf der Softwareseite. Unsere Forschung markiert den ersten Schritt hin zur Ausführung von Molflow-Simulationen auf GPUs.

Acknowledgements

I am deeply grateful to the individuals and groups who have played a significant role in my doctoral journey. Your support, guidance, and encouragement have been invaluable to me.

First and foremost, I extend my heartfelt gratitude to my university supervisor, Bruno Lang, for his guidance and support throughout my research. His expertise and insights have been instrumental in shaping this thesis. Likewise, I am also thankful to Peer Ueberholz, who not only served as a supervisor but also acted as a mentor throughout my university career. His wisdom and encouragement have been a constant source of inspiration.

To the TE-VSC group at CERN and its group leader, Paolo Chiggiato, I extend my appreciation for their support and collaboration. Paolo's dedication to fostering research excellence, particularly in nurturing the growth and success of PhD students, has greatly enriched my academic experience. Special thanks to my thesis supervisor at CERN, Roberto Kersevan, for his guidance, patience, and support throughout the development of this thesis. His provision of academic freedom was invaluable in shaping my research direction. Additionally, I acknowledge the contributions of my co-worker, Marton Ady, to the project.

To my friends and colleagues at CERN, thank you for your camaraderie, support, and stimulating and inspiring discussions we've shared during our coffee breaks. Your friendship has made this journey memorable and rewarding.

Lastly, I am indebted to my family and friends, especially my Mum, who has supported me unconditionally. Her sacrifices and unwavering belief in me have been my greatest source of strength.

To all those mentioned and those who may not be, I offer my heartfelt thanks. Your support has made this achievement possible.

Contents

Nomenclature

Acronyms

AABB	Axis Aligned Bounding Box
AC	Angular Coefficient
ADS	Acceleration Data Structure
ASCBR	Acceptable Shifting Convergence Band Rule
BLAS	Basic Linear Algebra Subprograms
BVH	Bounding Volume Hierarchies
CAD	Computer-Aided Design
CDF	cumulative distribution function
CFD	Computational Fluid Dynamics
CGAL	Computational Geometry Algorithms Library
CI/CD	Continuous integration and continuous delivery
CIs	Confidence Intervals
CLI	Command Line Interface
CLIC	Compact LInear Collider
CLT	Central Limit Theorem
COV	Coefficient Of Variation
CSG	Constructive solid geometry
DSMC	Direct Simulation Monte Carlo
GPGPU	General-Purpose Computing on Graphics Processing Units

GPUs	Graphics Processing Units
HEP	High Energy Physics
HPC	High Performance Computing
HRH	Hit Rate Heuristic
HV	High vacuum
i.i.d.	independent and identically distributed
LCG	Linear congruential generator
LHC	Large Hadron Collider
MIMD	Multiple Instructions Multiple Data
MSE	Mean Squared Error
NAO	Neighbour Aware Offset
NEG	Non-evaporable getter
PCG	Permuted Congruential Generator
PDF	Probability Density Function
PRNG	Pseudo Random Number Generator
PS	Progenitor Structure
PSD	Photon stimulated desorption
RF	radiofrequency
RT	Ray tracing
SAH	SURFACE AREA HEURISTIC
SIMD	Single Instruction Multiple Data
SIMT	Single Instruction, Multiple Threads
SM	Streaming Multiprocessors
SoA	Structure of Arrays
SR	Synchrotron Radiation

SRDH	Shadow Ray Distribution Heuristic
STL	Standard Triangle Language
STS	Specified Time Series
TPMC	Test Particle Monte Carlo
UHV	Ultra high vacuum
UX	User Experience
VSC	Vacuum, Surfaces, and Coatings
XHV	Extreme ultra high vacuum

Greek Symbols

α, β, γ	barycentric coordinates representing relative weights of triangle vertices in GPU intersection tests
δ	adaptive offset along the facet normal in GPU kernel trace processing
$\mathcal{G}$	geometry defining a polygon scene
Ω	set of polygons in a geometry
ϕ	azimuth angle used in spherical coordinate transformations in GPU kernel ray generation
τ	opacity value of a transparent facet
θ	polar angle in spherical coordinate transformations in GPU kernel ray generation

Latin Symbols

Δt_{max}	Maximum time step size in a sequence of time-dependent simulations
Δt_{min}	Minimum time step size in a sequence of time-dependent simulations
$\dot{n}_{\Sigma}$	total flux rate of the whole system in GPU simulations

$\mathbf{B}$	bitangent vector in 3D space
$\mathbf{I}$	all valid points on a ray
$\mathbf{M}$	transformation matrix for mapping global coordinates to local coordinate system
$\mathbf{N}$	surface normal in 3D space
$\mathbf{o}_{fac}$	origin of a facet's plane
$\mathbf{o}_{ray}$	origin of a ray
$\mathbf{p}$	all valid points on a plane
$\mathbf{p}_{local}$	local coordinates of a point in surface's 2D space
$\mathbf{r_{dir}}$	normalized direction of a ray
$\mathbf{T}$	tangent vector in 3D space
$\mathbf{u}, \mathbf{v}$	coordinate vectors defining a plane
$\Sigma I_{\perp}$	sum of orthogonal momentum changes of particles in GPU kernel statistics
$\Sigma v_{\perp}^{-1}$	sum of reciprocals of the orthogonal speed components of particles in GPU kernel statistics
$\textsc{rt}(\Omega, r)$	ray tracing query for geometry Ω and ray r
$\textsc{r}$	randomly generated number used in various calculations in GPU kernel development
$\textsc{x}, \textsc{y}$	uniformly distributed random numbers used in ray generation and direction calculation in GPU simulations
$\mathrm{d}t$	Time step size for dividing a time range into discrete moments in time-dependent simulations
A, B	minimal and maximal points defining an Axis Aligned Bounding Box (AABB)
dN_f/dt	local influx rate of particles for each facet in the GPU kernel
N	Number of discrete time moments in a specified time series for time-dependent simulations

N	number of polygons in set Ω
N_{hit}	number of collision events or Monte Carlo Hits in GPU kernel statistics
P_i	the i^{th} polygon in set Ω
p_Q	probability that a particular facet will be chosen as the starting location for particle outgassing in GPU simulations
r	ray, an oriented line
T	Tuple representing a specified time series (STS) in the form $t_{start}, t_{end}, t_w, \mathrm{d}t$ in time-dependent simulations
t	Time variable used in time-dependent simulations for tracking particle hits and system parameters
T_C	number of cycles for batched random number generation in GPU kernel development
t_{end}	Ending time for a specified time series (STS) in time-dependent simulations
t_{hit}	distance from ray's starting point to intersection point
t_{max}	Maximum time value in a simulation range for time-dependent analysis
t_{min}	Minimum time value in a simulation range for time-dependent analysis
t_{near}, t_{far}	values representing the intersection distances for AABB
t_{start}	Starting time for a specified time series (STS) in time-dependent simulations
t_w	Time window size for a time bin in time-dependent simulations
u_{hit}, v_{hit}	local 2D coordinates of intersected polygon

1 Introduction

Many scientific experiments such as the Large Hadron Collider (LHC) at CERN or the SPHEREx mission at NASA deploy critical vacuum components, where vacuum simulations are a key aspect in bringing these experiments to life. Molflow+ is a simulation software for ultra-high vacuum that deploys the Test Particle Monte Carlo (TPMC) method. The TPMC method is especially useful for modelling arbitrary complex vacuum components, as it converges sufficiently fast with accurate results. Each particle's trajectory and interactions within the vacuum system are simulated individually, providing a statistical approach to understanding the dynamics of ultra-high vacuum conditions. However, with the introduction of time-dependent simulations in Molflow+ (refer to Marton Ady, 2016), the demand for high-performing simulations has been rising. The demand for enhanced simulations is driven by the need to handle greater number of Monte Carlo events needed for detailed analysis and to make accurate assumptions about a given model. In addition, the ability to explore a wide range of simulation parameters, combined with the dependence on regular computing resources with lesser capabilities than high-end systems, underscores the need for stronger simulation solutions.

Developed originally by Roberto Kersevan in the 1990s, Molflow+'s development is now driven by CERN's VSC (Vacuum, Surfaces, and Coatings) group of the technology department, which is responsible for the design, construction, operation, maintenance, and upgrade of various vacuum components deployed at CERN, e.g. for the present and future accelerators and detectors. Alongside Molflow+, Synrad+ operates on a modification of the TPMC method (Roberto Kersevan and Marton Ady, 2019). It focuses on tracing photons instead of molecules for simulating Synchrotron Radiation (SR) effects, a phenomenon where accelerated charged particles emit electromagnetic radiation. This radiation is a critical factor in the design of particle accelerators and other high-energy physics environments, as it can cause significant heat load and gas desorption from exposed surfaces, which leads to instrumental damage in the worst case. Synrad+ allows to define magnetic regions where the trajectory of the beam is calculated, enabling the precise emission and interaction modelling of photons with surfaces.

Beyond the technical sphere, Molflow+ and Synrad+ significantly contribute to projects ranging from high-energy experiments such as particle accelerators (Kamiya et al., 2021; Roberto Kersevan, 2022) to space exploration (Alred et al., 2021; Ricchiuti, Fugett, and Soares, 2022) and laser technology (Song and Hernandez-Garcia, 2012; Aasman, 2020). Their extensive features make them already invaluable tools for physicists and engineers due for quickly analysing potential designs. However, the demand to design and investigate more variations of geometries with complex parameters calls for improvements in responsiveness and computational speed. It is not only on the side of the Monte Carlo simulation, but improvements on the pre-processing side will also lead to better usability and thus an improvement in productivity for the users.

A vacuum is a condition or space where the pressure is significantly lower than atmospheric pressure of 1013 mbar, achieved by removing air and other gases to create an environment with few or no particles. Such conditions are vital for the functionality and integrity of numerous applications. Take, for instance, CERN's LHC experiment which features a network of vacuum pipes spanning 104 km. Within this system, approximately 54 km of the pipes maintain a pressure in the order of 10^{-10} mbar to 10^{-11} mbar[1]. These vacuum levels are essential to minimize interferences that could compromise the precision and outcomes of the experiments conducted.

Designing vacuum components is often a trial-and-error process. In earlier design phases, where continuous feedback is desirable and queue times for HPC resources are often too long, physicists and engineers often utilize everyday computer hardware in the form of laptops or office computers, so running simulations on high-performance computers is not always an option. In later stages, the simulations shift to more powerful hardware, as improved convergence accuracy is necessary. Thus, making use of the available – also often affordable – hardware is desirable, where GPUs are always good candidates for computationally demanding tasks. Further, we will refer to Molflow+, when the C++ implementation for the CPU simulation engine is explicitly mentioned and to Molflow (without the suffix) when talking about general aspects of the algorithm.

1.1 Description of the problem

Molflow models are commonly imported from other CAD software, where surfaces are approximated with triangles and saved in STL format. The complexity of these models can vary greatly, where the amounts of surfaces and vertices have a direct impact on the simulation performance. In Molflow, the heart of these simulations lies in the ray tracing engine. It is a critical component responsible for accurately simulating particle trajectories and interactions within the modelled vacuum system. The performance of this engine is a key component for Molflow, as it directly influences the efficiency and effectiveness of the entire simulation and therefore the

[1]Accessed on 23/11/2023:
`https://home.cern/science/engineering/vacuum-empty-interstellar-space`

design process. Engineers involved in vacuum simulations value efficiency at all stages. This includes the pre-processing, where imported models are modified and adjusted, the core simulation phase, driven by the ray tracing engine, and the post-processing phase, where simulation results are evaluated in real-time and after a simulation concludes.

Our initial analysis, detailed in chapter 3.6, involved benchmarking the performance of various routines across all these stages, identifying potential bottlenecks, particularly in the ray tracing algorithm, that could be optimized. We found that for most models the ray tracing routines contribute to around 80% of the computation time. In addition, we identified a potential bottleneck attributed to time-dependent simulations. Further, with an investigation on the whole workflow, discrepancies on the performance of some pre-processing steps were found. In particular, the analysis of the angular relations of neighboring surfaces in a mesh could potentially take several minutes and in more extreme cases even hours.

1.2 Motivation of this thesis

This thesis examines a new approach in the simulation processes of Molflow and Synrad. The primary objective is to explore and implement advanced computational techniques, with a specific focus on refining data structures and investigating the potential of GPU simulations. This is aimed at significantly improving the efficiency and accuracy of the simulation engine. The algorithmic backend for pre- and post-processing routines is mainly driven by algorithms dealing with the analysis and modification of 3D geometry. We aim to improve the simulation engine performance by optimizing and developing specialized data structures and routines for the ray tracing algorithm. By exploring the parallel computing capabilities, our aim is to enable the simulation environment to operate effectively in a High Performance Computing (HPC) environment, leveraging both modern CPU architectures and GPUs. With a focus on the proven CPU simulations, this involves developing new ray-tracing techniques specifically tailored for Molflow's use case and comparing their performance against state-of-the-art techniques commonly employed in other applications, especially in graphical rendering. With the advent of dedicated ray tracing units in modern GPUs, the promise for major performance improvements is encouraging the reevaluation of a GPU kernel for Molflow. Within two generations, NVIDIA promises ray tracing performance increases of $\sim 10\times$.

A significant part of this work involves evaluating the suitability of Molflow's algorithm for GPU processing, particularly with the use of novel ray-tracing cores found in NVIDIA's RTX GPU series. A crucial step in achieving this incorporates a major revision of the software development process and the underlying architecture that lead to starting with the development of a Command Line Interface (CLI). Not only, does the CLI enable users with swift simulation launches, but it also paves the way into a new testing infrastructure. By enhancing the capabilities of Molflow+ and Synrad+, this thesis directly contributes to the efficiency and accuracy of simulations in groundbreaking projects in various fields beyond CERN's particle accelerators.

1.3 Outline of this thesis

This thesis is structured as follows to comprehensively cover the various aspects of improving Molflow's simulation engine and its application in vacuum simulations: We begin this thesis in chapter 2, by elaborating the foundational concepts of vacuum physics, the principles of the TPMC method, and the specifics of Molflow and Synrad. This chapter also explores the fundamentals of ray tracing, and various case studies showcasing real-world applications for the simulations. This is followed in chapter 3 with a discussion about the methodologies employed in refining the software, including code refactoring, the development of a command line interface, continuous integration and deployment with GitLab with an automatic testing environment, and compatibility across different operating systems and compilers. An initial profiling to identify and address performance bottlenecks is also covered. In chapter 4, the focus is on the characteristics of different geometries used in simulations and the impact they have on the performance and accuracy of Molflow and Synrad. Chapter 5 delves into the software architecture, representation of geometry within the software, and the advancements in time-dependent simulations. In addition, it details the challenges and solutions in improving the mesh analysis of neighbourhood relationships. The motivation of iterative algorithms for simulations is explored in chapter 6. This includes an implementation and analysis of convergence criteria. Chapter 7 gives an in-depth look at the development and application of advanced ray tracing techniques. This includes splitting heuristics tailored toward Molflow's use case for the construction of bounding volume hierarchies and KD-trees. Benchmarks and evaluations of these techniques against state-of-the-art techniques are presented. In chapter 8, we revisit the suitability of Molflow's algorithm for GPU processing by evaluating the use of novel ray-tracing cores for NVIDIA RTX GPUs. After exploring the basics of GPU computing, the challenges in GPU ray tracing, and the development of the GPU kernel with NVIDIA's OptiX API are discussed. A performance and precision study of the GPU implementation is also included. Lastly, the final chapter 9 summarizes the key findings and contributions of the thesis. It also outlines potential future research directions and the long-term implications of this work for vacuum simulation studies.

2 Background

In this chapter the general concept of the Molflow and Synrad simulation software will be elaborated. First, a general explanation for vacuum simulations and an overview of common vacuum simulation methods and tools is given. Next, the algorithm and statistical model for Molflow's simulation engine is explained. Furthermore a brief explanation about Ray Tracing and various acceleration techniques will be given and discussed in more detail in the context of Molflow.

2.1 Vacuum

In this chapter, we will give a brief explanation on fundamental concepts in vacuum physics necessary for a better understanding of key concepts in vacuum simulations.

Vacuum refers to a space devoid of matter, a concept that, in practice can only be approximated. In more realistic terms, we describe a partial vacuum as a space where the pressure is much lower than that of the Earth's atmosphere. Pressure, defined as the force exerted on an object per unit area, is typically expressed as:

$$p = \frac{F}{A}\,, \tag{2.1}$$

with the pressure p, the force F and the area A. The standard unit of measurement for pressure is Pascal (Pa), defined as $1\text{Pa} = 1\text{N/m}^2$. Alternatively, pressure can also be measured in millibars (mbar), where $1\text{mbar} = 100\text{Pa}$. Furthermore, there are other units such as $1\text{Torr} \approx 133.32\text{Pa}$. Depending on the pressure within a space, we classify vacuums into different types[1]:

- **Low vacuum:** Pressure ranging from 1 to 1013.25 mbar.
- **Medium vacuum:** Pressure between 10^{-3} and 1 mbar.
- **High vacuum (HV):** Pressure within the range of 10^{-7} to 10^{-3} mbar.
- **Ultra high vacuum (UHV):** Pressure ranging from 10^{-11} to 10^{-7} mbar.
- **Extreme ultra high vacuum (XHV):** Pressure below 10^{-11} mbar.

[1]Accessed on 05/09/2023: `https://www.pfeiffer-vacuum.com/en/know-how/introduction-to-vacuum-technology/fundamentals/definition-of-vacuum/`

Here are some practical examples to illustrate these vacuum levels:

- **Light Bulb:** The pressure inside a standard incandescent light bulb is typically around 700 mbar. This partial vacuum prevents the filament from oxidizing and burning out.
- **Vacuum Cleaner:** Household vacuum cleaners generate a suction power of approximately 100 mbar to effectively pick up dust and debris from floors.
- **Semiconductor Manufacturing:** Vacuum is crucial in semiconductor fabrication to ensure precise material deposition without contamination, where some processes operate at 0.1 mbar.
- **Large Hadron Collider (LHC):** The vacuum level within the LHC vacuum pipes is maintained at an extraordinary range of 10^{-10} to 10^{-11} mbar to reduce unwanted particle interactions during high-energy collisions.
- **Quantum Computing:** Experiments in quantum computing often require vacuum states approaching 10^{-12} mbar to isolate qubits from external interference.
- **Space Pressure:** In outer space, the pressure is extremely low, close to a hard vacuum with pressures below 10^{-12} mbar. This vacuum environment is a critical consideration for spacecraft design and space exploration.

In order to create and maintain certain pressure levels in experimental conditions, different types of vacuum pumps are used depending on the requirements. To achieve the vacuum levels found in the beam pipes and detectors of the Large Hadron Collider (LHC) at CERN different pumps are used. To give some examples:

- **Turbomolecular Pumps:** Turbomolecular pumps are essential in the LHC for achieving and maintaining high vacuum levels. These pumps use a series of high-speed rotating blades to propel gas molecules out of the vacuum chamber. Turbomolecular pumps are known for their efficiency and are crucial for creating the extreme vacuum conditions required in the LHC.
- **Ion Pumps:** Ion pumps, also known as Penning traps, are used to maintain ultra-high vacuum conditions in specific sections of the LHC. They work by creating and trapping ions within the vacuum chamber, effectively removing gas molecules. Ion pumps are capable of achieving extremely low pressures and are suitable for UHV and XHV applications.
- **Getter Pumps:** Getter pumps are used in some sections of the LHC to maintain low pressures. They work by adsorbing and chemically binding residual gas molecules, helping to achieve and maintain vacuum conditions.

These and other types of vacuum pumps, often used in combination, play a critical role in creating and maintaining the ultra-high vacuum conditions necessary for the LHC's particle collision experiments. Each pump type is strategically employed in different sections of the accelerator to meet the specific vacuum requirements of that area.

Depending on the pressure level, the behaviour of a gas is different. This behaviour can be classified based on the Knudsen number:

$$Kn = \frac{\lambda}{L}\,, \tag{2.2}$$

where λ is the mean free path and L is the characteristic length of the geometry. The characteristic length describes the length of the dimension that alone can describe the molecular dynamics best. The mean free path can be described as the average distance a particle travels in a system that changes neither its direction nor its energy significantly. This may happen due to collisions with other particles or boundary walls. It can be calculated by

$$\lambda = \frac{k_B T}{\sqrt{2}\pi d^2 p}\,, \tag{2.3}$$

with the Boltzmann constant $k_B[J/K]$, the absolute temperature $T[K]$, the gas pressure $p[Pa]$ and the gas particle diameter $d[m]$. Given the Knudsen number, we categorise the molecular behaviour with the following three regimes:

- Continuous flow $Kn < 0.001$: Frequent intermolecular collisions, less frequent wall collisions. Gas behaviour can be described as a whole with a set of governing equations, often approximately solved in the field of Computational Fluid Dynamics (CFD).

- Knudsen flow $0.001 < Kn < 1$: Transitional regime, where intermolecular collisions still contribute to the gas behaviour.

- Free molecular flow $1 < Kn$: Molecules move quasi-independently from each other with little to no intermolecular collisions.

For free molecular flow, the condition that is typically achieved in particle accelerators as it is favourable in reducing disturbances, there exists no set of governing equations, which we can integrate and solve to get the state in space and time dimensions. The gas behaviour has to be modelled statistically, where a large number of particles are treated independently. For less complex ultra high vacuum problems, analytical solutions exist to calculate some characteristics of a vacuum chamber. The fundamental equations to calculate the equilibrium pressure $p\,[mbar]$ and the conductance $C\,[l/s]$ for a vacuum component are:

$$p = \frac{Q}{S}\,, \tag{2.4}$$

$$C = \frac{Q}{dp}\,, \tag{2.5}$$

where $Q\,[mbar \cdot l/s]$ is the gas inflow rate, $S\,[l/s]$ is the pumping speed, and dp is the pressure difference. Conductance C is the gas flow rate divided by the pressure drop between two points under steady conditions, it is relevant to understand the gas transmission between different regions of a system. The pressure p for a vacuum component is given by the gas flow rate divided by the pumping speed S, which is defined by how much volume is pumped per unit time. In figure 2.1 this is shown for a system in steady-state. From the left side, a gas inflow Q leads to a first chamber with measured pressure p_1. The pressure in the second chamber is measured with p_2, where $p_1 > p_2$. Between the two chambers, the conductance is calculated by $C = \frac{Q}{p_1 - p_2}$. On the right of the schematic, the gas is pumped with the pumping speed $S = \frac{Q}{p_2}$.

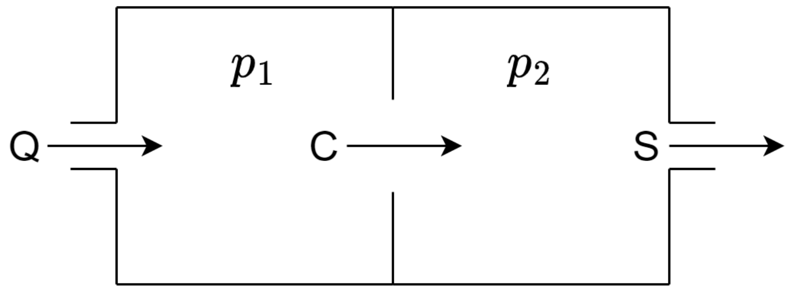


Figure 2.1: Schematic of a simple vacuum problem with an inlet an outlet and two connected chambers with different pressures. The system is in steady-state.

Analytical solutions can be practically applied for structures that can expressed as a linear series of vacuum components for which the conductance is known for all elements. Such methods are used widely in several codes at CERN, such as VASCO (Rossi, 2004). Unfortunately, finding analytical solutions of the conductance for all kinds of geometries is nearly impossible. For that reason, the characteristics of vacuum components are usually calculated approximately with Monte Carlo methods.

2.2 Monte Carlo simulations with Molflow

This chapter first elaborates on the physical model and algorithm behind Molflow and Synrad, based on the work of Roberto Kersevan and Pons (2009) and Marton

Ady (2016). In addition, general use cases of both applications are introduced, and alternative software solutions and algorithms are discussed. Our focus is to outline the individual components of Molflow's Monte Carlo model, laying the groundwork for subsequent in-depth analyses aimed at identifying and implementing algorithmic optimizations and improved data structures.

2.2.1 Motivation

Higher energy in modern accelerators increases the amount of radiation, which leads to the desorption of photoelectrons and molecules from the accelerator walls. This promotes electron cloud buildup and an increase in residual pressure, which are the limiting factor to the beam lifetime.

Various methods and tools exist to help with the calculation of synchrotron radiation. Molflow and Synrad are a set of such tools, where one is designed for the calculation of pressure profiles and the other for the calculation of synchrotron radiation. The tools are based on the Monte Carlo method, where the path of uniformly distributed molecules, respectively, photons is traced and their interactions with the geometries' surfaces are used to calculate the physical quantities.

In accelerators, one is typically dealing with a high vacuum or ultra-high vacuum, where the gas dynamics are described by the free molecular flow regime ($1 < Kn$). Analytical models exist, but they have the fundamental drawback that it's not easy to apply them to complex geometrical structures as they are mostly dependent on analytic expressions for the conductance, which in general is the unknown one is interested in (see chapter 2.1). Molecules have to be treated individually in this regime, and thus one cannot simply describe them by a set of differential equations like in fluid dynamics. Various methods exist to approximate characteristic values. Most prominently the Direct Simulation Monte Carlo (DSMC) method, the view factor method and Test Particle Monte Carlo method, which is applied in Molflow and Synrad.

The angular coefficient method (Saksaganskii, 1980), also known as view factor method, allows the computation of various characteristic values by approximating how well one surface can see another surface. View factors $F_{i,j} \in [0, 1]$ are calculated based on the area, distance and angle between the surface normals of two elements i, j from the discretized surfaces of the input geometry. Further, based on the view factors $F_{i,j}$ the flux of incoming particles on a surface i from the outgoing particles of a surface j can be calculated. When discretizing a geometry into N elements, a view factor matrix of $N \times N$ is created, hence, one has to often sacrifice accuracy to compute a timely solution by approximating the geometry only by a minimal amount of surface elements. The view-factor method can further be extended for time-dependent simulations by calculating the corresponding view factors with discrete time steps (Drake, 1990). Only the flux contribution during a time step will be used to compute the time-dependent state of an element. The method is derived from the radiosity method, which has been originally used for applications such as heat-transfer simulations, and has been further refined for the use in computer graphics, vacuum simulations and other disciplines. Due to the large demands on memory, the method is mainly suitable for calculations on computing clusters.

The DSMC method (Bird, 1976; Hasan, 2020) divides a given geometry into computational cells. For every iteration, inserted particles are then advanced in time. First, particles are advanced directly in space along their trajectory. Next, particles are randomly chosen to calculate intermolecular collisions to gather further statistics to model effects in the viscous regime ($Kn < 1$). Depending on the space and time discretization, the DSMC method is both expensive on the memory and the computational demand, when compared to other methods, in particular, the TPMC method applied in Molflow.

2.2.2 The Test Particle Monte Carlo method

Molflow is a simulation software for ultra-high vacuum, which is built around a Monte Carlo algorithm developed by Kersevan (1991), the so-called Test Particle Monte Carlo (TPMC) algorithm. Test particles are generated from facets, which are used to describe the geometry, where an influx of gas molecules into the system is defined. Possible starting points are uniformly distributed across all of these source facets and one is chosen randomly. In the simulation, each particle is assigned a direction and speed through random selection. The direction is determined using Knudsen's cosine law, a principle that predicts how particles scatter, with randomness introduced for variability. The speed is derived from the Maxwell-Boltzmann distribution, a fundamental concept in physics that describes how the speed of particles in a gas varies. Afterwards, a ray tracing algorithm is used to calculate the next collision point with a facet from the geometry. Collision events and the corresponding hit locations are then used to gather statistics, which can then be used to calculate characteristic values of the geometry.

The algorithms behind Molflow and Synrad (see Marton Ady, 2016, and Roberto Kersevan and Marton Ady, 2019) follow the same principles, but for simplicity, the explanation is done with Molflow as the main code in mind. The algorithm's steps are depicted in figure 2.2 and are elaborated in more detail in the following chapter.

2.2.3 The Monte Carlo model

Molflow follows the idea of a Monte Carlo simulation, where the desired values are expressed as the mean of a random variable $\hat{\mu}$. By repeating the same experiment multiple times we get a sample size n, from which we can approximate the mean of

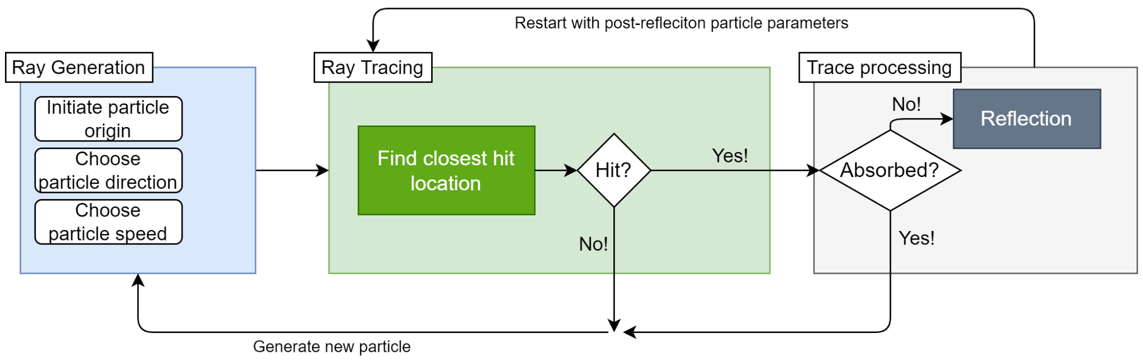


Figure 2.2: Sketch of the pipeline for Molflow's Monte Carlo algorithm. Three-step approach starting with Particle generation, a ray tracing routine and lastly trace processing that takes care of statistic gathering and setting up the particle state for the next iteration.

the specific values. A larger sample size leads to sufficiently low errors from the real mean μ, where for $n \to \infty$ the approximated mean approaches the real mean with $\lim_{n\to\infty} \hat{\mu} = \mu$. T. Booth (2015) elaborates on the foundations and problems to avoid for the design of MC simulations in the context of particle transport.

Various traits of free-molecular flow can be described as random processes. Given a probability function for such a process, a sample can be selected by using a random number to simulate an event. Following, the individual parts of the Monte Carlo simulation are elaborated including their probability distributions.

With the TPMC method, a large number of particles is represented by a smaller number of test particles N_{test}. Here the ratio

$$K_{\frac{real}{test}} = \frac{\sum_f dN_{f,real}/dt}{N_{test}} \tag{2.6}$$

represents the total number of real particles $N_{f,real}$ per second entering from a facet f represented by a virtual test particle. Hence, a virtual particle represents a flux of real particles entering the system.

Starting parameters

We describe a particle's state in Cartesian coordinates by the tuple $\{\mathbf{o}, \mathbf{r}, v\}$, the origin in three-dimensional space, the particle direction as a three-dimensional vector and the particle velocity respectively.

Particle Origin: Molecules enter the system via injection or thermal desorption from the surfaces of vacuum components. This is modelled with the outgassing rate

Q_f (in $mbar \cdot l/s$), which represents the gas inflow from a given facet f. This outgassing rate can be expressed as the rate of change of the pressure-volume product:

$$Q_f = \frac{d(pV)}{dt}\,, \tag{2.7}$$

where p is the pressure (in mbar), V is the volume (in l), and t is time. We can then connect this outgassing rate to the ideal gas law, which describes the behavior of an ideal gas:

$$pV = N_{real} k_B T\,, \tag{2.8}$$

where N_{real} is the number of particles in the gas, k_B is the Boltzmann constant ($k_B \approx 8.3145$ in $\frac{J}{mol \cdot K}$), and T is the absolute temperature of the gas (in K). Hence, by applying (2.8) to (2.7), we can derive the influx of physical particles $\dot{n}_f$ for a facet f:

$$\dot{n}_f = \frac{dN_{f,real}}{dt} = \frac{d(pV)}{dt}\frac{1}{k_B T_f} = \frac{Q_f}{k_B T_f}\,, \tag{2.9}$$

where T_f is the average temperature corresponding to the facet f. A facet can then be sampled by randomly picking a facet $f_o \in \mathcal{Q}$ from a list of all desorbing facets $\mathcal{Q} = \{\forall f \in \mathbf{F} : Q_f > 0\}$, where each facet has a probability of $p_{Q_f} = \dot{n}_f / \dot{n}_\Sigma$ to get selected. The probability is simply the ratio of the local influx rate of the facet and the total influx rate of the entire system

$$\dot{n}_\Sigma = \sum_{i=1}^{N_f} \dot{n}_i\,, \tag{2.10}$$

which is the sum of the influx rates of all N_f facets. This approach ensures that facets with a higher local influx rate have a proportionately higher probability of being selected, reflecting their relative contribution to the total influx rate of the system. Using the cumulative distribution function of the artificial influx probabilities, one can then pick a random outgassing facet f_o with a random number $\mathrm{X} \in [0, 1]$. A simplified pseudocode iterative algorithm is given with algorithm 1. In the algorithm, we initialize a cumulative probability C to zero to keep track of the cumulative probability as we iterate through the facets[2]. For each facet f_i in the set of all desorbing facets $\mathcal{Q}$, we calculate the probability of selecting facet f_i as $p_{Q_i} = \dot{n}_i / \dot{n}_\Sigma$, where $\dot{n}_i$ is the influx rate for facet f_i and $\dot{n}_\Sigma$ is the total influx rate for all facets. After updating the cumulative probability: $C = C + p_{Q_i}$, we can check if C has reached or exceeded X. If that is the case, we select facet f_i as f_o and terminate the loop, otherwise we continue to the next facet.

[2]If cumulative probabilities are precalculated, we can use binary search to reduce the complexity to $O(\log N)$ for N facets. A sequential lookup $O(N)$ is given for simplicity. This is the deployed strategy for the newly developed GPU kernel, elaborated in chapter 8.

Algorithm 1 Select Random Outgassing Facet

```
Require: Random number $\textsc{x}$, Set of desorbing facets $\mathcal{Q}$, Total influx rate $\dot{n}_\Sigma$
1: Initialize cumulative probability $C \leftarrow 0$
2: for each facet $f_i$ in $\mathcal{Q}$ do
3:     Calculate facet's probability $p_{Q_i} \leftarrow \frac{\dot{n}_i}{\dot{n}_\Sigma}$
4:     Update cumulative probability $C \leftarrow C + p_{Q_i}$
5:     if $C \geq \textsc{x}$ then
6:         Select facet $f_i$ as $f_o$
7:         break
8:     end if
9: end for
```

Picking a random position **o** on the selected facet f_o is not as straightforward, because Molflow is using n-polygons to describe a geometry. Using local 2D coordinates to describe the position on the facet's plane, one can easily find a position u, v with two random numbers. Given the position u, v, one still has to check whether the point is lying inside the n-polygon or not. A point-in-polygon algorithm is proposed in chapter 2.3.5, which works with both convex and concave polygons and is independent with respect to orientation. The downside of such a method is that points can be generated that lie outside of the actual polygon. In such a case, new random numbers have to be generated to check for a new sample point. In practice, consecutive discards of sample points are relatively rare and don't have a big impact on performance. A more elegant solution is proposed as part of the GPU kernel (see chapters 8.3.2.7) to guarantee the generation of a sample point solely with two random numbers.

Particle Trajectory: We calculate the particle direction according to Knudsen's cosine law (Knudsen, 1967), which assumes that a particle's inbound and outbound direction are independent. It states that a particle leaving a surface in solid angle $d\omega$ is proportional to $\cos\theta$, with θ forming the angle between the surface normal and a particle's trajectory. The cosine law can be written as

$$\frac{1}{N_r}\frac{dn_r}{d\omega} = \frac{\cos\theta}{\pi}\,, \tag{2.11}$$

$$d\omega = \sin\theta d\theta d\phi\,, \tag{2.12}$$

with the total number of reflected particles N_r and dn_r, the number of particles emitted in solid angle $d\omega$. One can derive the azimuth and polar angles (ϕ respectively θ) as follows:

$$\phi = \textsc{x}2\pi \quad , \textsc{x} \in [0,1]\,, \tag{2.13}$$

$$\theta = \sin^{-1}\sqrt{\textsc{y}} \quad , \textsc{y} \in [0,1]\,, \tag{2.14}$$

where X and Y are uniformly distributed random numbers (Suetsugu, 1996). For use in a 3-dimensional Cartesian system, one has to transform the derived spherical coordinates further into Cartesian coordinates $(1, \theta, \phi) \rightarrow (x, y, z)$. First we derive the local Cartesian coordinates (u, v, n):

$$\begin{aligned} u &= \sin(\theta) \cdot \cos(\phi)\,, && (2.15)\\ v &= \sin(\theta) \cdot \sin(\phi)\,, && (2.16)\\ n &= \cos(\theta)\,, && (2.17) \end{aligned}$$

and translate them with the corresponding facet's orthonormal basis $\{\mathbf{U}, \mathbf{V}, \mathbf{N}\}$ into global Cartesian coordinates (x, y, z) for the particle direction:

$$\begin{aligned} x &= u \cdot U_x + v \cdot V_x + n \cdot N_x\,, && (2.18)\\ y &= u \cdot U_y + v \cdot V_y + n \cdot N_y\,, && (2.19)\\ z &= u \cdot U_z + v \cdot V_z + n \cdot N_z\,. && (2.20) \end{aligned}$$

The orthonormal basis in $\mathbf{R}^3$ consists of three vectors of length 1, which are all perpendicular to each other. In Molflow it is formed by $\mathbf{N}$, which is the facet normal, and the vectors $\mathbf{U}$ and $\mathbf{V}$, which form the local 2d coordinate system.

Particle Velocity: There are different methods to obtain a reasonable particle velocity for vacuum simulations. Assuming that an ideal gas is at equilibrium in a defined environment, we can describe the speed of molecules with the Maxwell-Boltzmann distribution. For simplicity, we can assume that the average velocity is sufficient to describe most effects inside a vacuum component. In case of the Maxwell-Boltzmann distribution it is given by:

$$< v >= \sqrt{\frac{8RT}{\pi m}}\,, \qquad (2.21)$$

where R is the ideal gas constant, T the gas temperature and m the molar mass (in kg). Given an isothermal system, one can further assume a constant velocity that is not affected by collisions.

The preferred method in Molflow, in more detail discussed by Marton Ady (2016), accounts for the effect of faster molecules hitting surfaces more often. Particles obtain a new speed after every collision with a wall, due to thermalisation effects, based on a facet's temperature T, that are happening while particles spend a sojourn time on the wall. We retrieve the new speed from the Probability Density Function (PDF) of molecules colliding with a wall during a period of time $f(v)_{coll}$[3]:

$$f(v)_{coll} = v^3 \exp(-\frac{v^2}{2a^2})\frac{1}{2a^4}\,, \qquad (2.23)$$

[3]This function can be derived from the PDF of molecular speeds, following the Maxwell-

with the scale parameter:

$$a = \sqrt{\frac{k_B T}{m}} .$$

Given the PDF, we can then calculate the cumulative distribution function (CDF), which for every speed value v assigns a probability that a particle's velocity is lower than v:

$$F(v)_{coll} = \int_0^v f(v')_{coll} dv' = 1 - \exp\left(-\frac{v^2}{2a^2}\right)\left[1 + \frac{v^2}{2a^2}\right] . \tag{2.24}$$

By generating CDFs for each temperature value T in the system, we can easily find a random speed value given a random number by numerical inversion. Marton Ady (2016) optimized this by using precalculated bins for equidistant speed values $v \in \left[0 \quad , \quad 4 \cdot \sqrt{\frac{2RT}{M_{[kg/mole]}}}\right]$ for every temperature value T in the system, where $\sqrt{\frac{2RT}{M_{[kg/mole]}}}$ represents the most probable speed value.

We generate a CDF corresponding to a particular temperature T for $N = 100$ bins. Each bin $i \in [0, N-1]$ corresponds to a tuple $\{v_i, p_i\}$. Here, v_i is the value

$$v_i = \frac{i}{N} \cdot 4 \cdot \sqrt{\frac{2RT}{M_{[kg/mole]}}} , \tag{2.25}$$

and $p_i \in [0, 1]$ is the probability value

$$p_i = F(v)_{coll} = 1 - \exp\left(-\frac{v^2}{2a^2}\right)\left[1 + \frac{v^2}{2a^2}\right] . \tag{2.26}$$

We can then find a random velocity value v_{R} using linear interpolation. First, given a random value $\mathrm{R} \in [0, 1]$ we identify the corresponding bin j via binary search on the probability values p_i:

$$\forall i < j , \quad \mathrm{R} > p_i \quad \text{and} \quad \mathrm{R} \leq p_j . \tag{2.27}$$

Here, j is the smallest index such that $\mathrm{R} \leq p_j$. Further, we require that j lies in the interval $[0, N-2]$. We can then calculate a velocity value via linear interpolation for the bins j and $j+1$:

$$\Delta p = \quad p_{j+1} - p_j , \tag{2.28}$$

$$\epsilon = \quad \mathrm{R} - p_j , \tag{2.29}$$

$$v_{\mathrm{R}} = \quad v_j + (v_{j+1} - v_j)\frac{\epsilon}{\Delta p} . \tag{2.30}$$

Boltzmann distribution, in a volume $f(v)_{gas}$:

$$f(v)_{gas} = v^2 \exp(-\frac{v^2}{2a^2})\frac{1}{a^3}\sqrt{\frac{2}{\pi}} . \tag{2.22}$$

See Marton Ady (2016) for proof.

Here, Δp defines the difference between the two probability values and ϵ represents how much the random value R exceeds the probability value of the lower bound bin j.

Next location

Molflow is using a ray tracing algorithm (see chapter 2.3.3) to find the collision location given a particle and its trajectory. First, given the ray tuple $\{\mathbf{o}, \mathbf{r}, v\}$ the algorithm will return the closest hit location as a pair of facet ID and the intersection location in 3d space coordinates. Next, depending on the type of facet with which the particle is colliding, different cases have to be considered. Most importantly, these are solid and transparent surfaces, and linked or teleport facets, which have been sketched in figure 2.3.

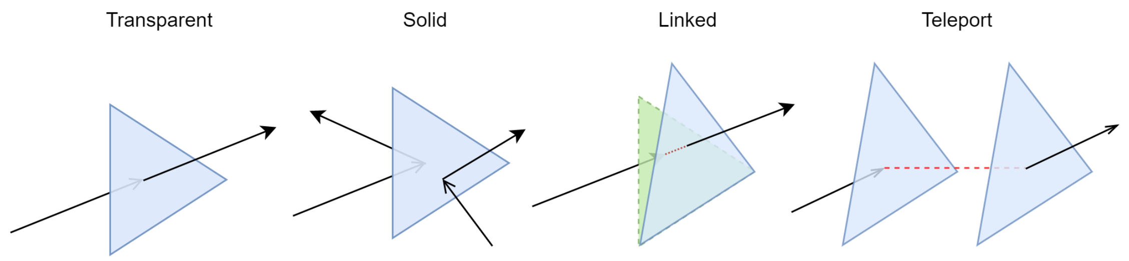


Figure 2.3: Sketch of facet types in Molflow. Rays pass through transparent facets without changing their direction. Rays colliding with solid facets have a direction change. Linked facets are two separate facets (here, green and blue) that are linked together, creating an entry and an exit for different structures. Two teleport facets are linked together. On collision, a ray is transported to the opposite one without changing the direction.

Solid facet: When a particle collides with a solid facet, it will always be reflected. The reflection occurs according to Knudsen's law (2.11). A solid facet can be modified with desorbing properties, with an outgassing rate (2.7), or absorbing properties, which are elaborated next.

Absorbing facet: A hit with an absorbing facet can lead to a particle either bouncing off or getting absorbed. This can be described by a facet's sticking factor $s \in [0, 1]$, from which the pumping speed S $[m^3/s]$ can be derived:

$$S = s \cdot \frac{1}{4} < v > \cdot A \,, \tag{2.31}$$

where A is the surface area of the corresponding facet (in m^2) and $< v >$ the average speed of a particle (in m/s), which in case of the Maxwell-Boltzmann distribution is given by (2.21). A sticking factor $s = 0$ means that a particle will definitely bounce off (a perfect solid facet) and a value of $s = 1$ implies that a particle will be removed from the system. Given a randomly generated number $\text{R} \in [0, 1]$, we can determine directly from the sticking factor, whether a particle rebounds or gets pumped. A particle is absorbed, if $\text{R} \leq s$.

Transparent facet: Transparent facets have an opacity value $o \in [0, 1]$ that denotes the probability that a particle is passing through. This is useful e.g. for modelling holes on a facet without an explicit geometric description, or to create facets to visualise physical values in the middle of a volume. Therefore, the ray tracer has to determine, given a random value and the facet's opacity value, whether the particle will pass through or bounce off a transparent facet. Practically, in Molflow, transparent facets are handled during the ray tracing procedure (see chapter 2.3.1). A collection of passed transparent facets is evaluated in the trace processing step, by incrementing the appropriate counters.

Link facet: Further, a geometry can be divided into multiple structures, which are then connected by two quasi-identical link facets: one for each structure. The ray tracer will only consider potential collisions for facets within the residing structure. Only if a link facet is crossed, the corresponding facets will be taken into consideration during a second ray tracing iteration.

Teleport facet: Similar behaviour to link facets is achieved by teleport facets. A collision with such a facet teleports the particle to another facet with the same shape, but at a different location. This enables designers to easily construct repeating geometries. In case of a collision, the particle is properly translated and rotated, and further handled in a second iteration step. For more details, we redirect the readers to the work of Marton Ady (2016).

Trace processing: Hit counter

When the new particle position and state has been decided, statistics are gathered on the intersected facet. Molflow has different methods to gather these statistics depending on the necessary granularity. In general there are three different counters that are incremented with every collision.

Number of hits N_{hit} is a simple counter that accumulates the number of Monte Carlo hits. Each time a particle collides with the facet, this counter is incremented. From the N_{hit}, we can derive the impingement rate (2.38).

Total orthogonal momentum change $\Sigma I_\perp$ is a counter that increases with the orthogonal momentum change of the incoming and outgoing particles: $mv_\perp = mv\cos\theta$. Here, θ is the incident angle at which the particle collides with the wall, m is its mass, and v is the velocity of the particle. $\Sigma I_\perp$ is used to compute the pressure (2.39) exerted by particles on the facet. It accounts for the momentum transfer in the direction perpendicular to the facet, reflecting the force exerted by the particles.

Sum of reciprocals of orthogonal speed components $v_\perp^{-1}$ is a counter that increments the reciprocal of the orthogonal speed components: $v_\perp^{-1} = 1/v\cos\theta$, where $v\cos\theta$ represents the orthogonal component of the particle's velocity. $v_\perp^{-1}$ provides insight into the time particles spend in close proximity to the facet. It can be used to retrieve the particle density (2.40).

Additional counters: Additional counters exist, which we give for the sake of completeness. The number of absorptions N_{abs} and the number of desorptions N_{des} are incremental counters. The sum of reciprocals of velocity components v^{-1} is a counter that accumulates the reciprocals of the velocity of each particle. For each particle interaction, it increments by the value $\frac{1}{v}$, where v is the particle's velocity. This accumulation is particularly useful for calculating the average molecule speed in the simulation based on the harmonic mean.

Trace processing: Textures and profiles

A more precise way to collect statistics in Molflow is the deployment of the so-called *profiles* and *textures*. They are essential for use cases where detailed spatial distribution of particle interactions is crucial for analysis and design.

Profiles Profiles are used to divide a facet in one direction. Along one of the axes of the local 2d coordinates u, v with $u, v \in [0, 1]$, a profile creates P equidistant slices, typically $P = 100$. Given a particle hit location $\mathbf{p} = (p_u, p_v)$ on facet f, where p_u and p_v are the coordinates along the local u, v coordinates of f, then for a profile along the u direction we can calculate the index of the slice with:

$$\mathrm{pos}_U = \lfloor p_u \cdot P \rfloor \ . \tag{2.32}$$

For a profile along the v direction we can calculate the index of the slice with:

$$\mathrm{pos}_V = \lfloor p_v \cdot P \rfloor \ . \tag{2.33}$$

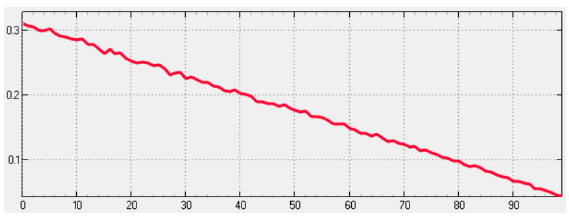
Here, $\mathrm{pos}_U, \mathrm{pos}_V \in [0, P-1]$, where $\mathrm{pos}_U, \mathrm{pos}_V \in \mathbb{N}^+$. Profiles allow to collect the number of hits, the momentum changes ($\Sigma I_\perp$), and the sum of reciprocals of orthogonal speed components of particles ($\Sigma v_\perp^{-1}$). In addition, there is a special type of profile to analyze the distribution related a particle's incident angle on a facet f. An angular profile creates P slices along the angular range $[\pi/2, \pi]$, where each size of each angular slice is given by $\Delta\theta = \frac{\pi}{2}/P$. Given a particle's direction vector $\mathbf{d}$ and the normal vector $\mathbf{N_f}$ of the facet f, we can calculate the incident angle θ with:

$$\theta = \arccos(|\mathbf{N_f} \cdot \mathbf{d}|)\,, \tag{2.34}$$

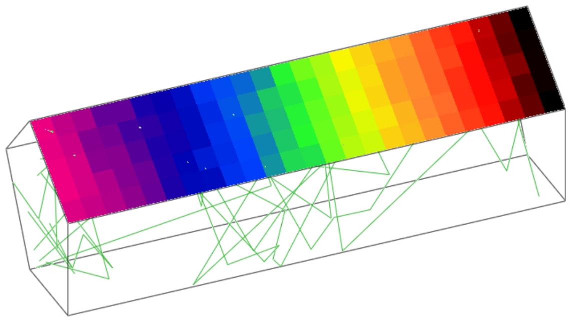
and the corresponding bin index on the angular facet with:

$$\text{pos}_\theta = \left\lfloor \frac{\theta}{\Delta\theta} \right\rfloor = \left\lfloor \frac{2\theta}{\pi} \cdot P \right\rfloor\,. \tag{2.35}$$

An example showing a pressure profile on an arbitrary Molflow geometry is shown in sketch 2.4a. Here, a profile is deployed along the u direction of the facet (here, it is the long side). It is deployed to count the sum of reciprocals of orthogonal speed components $v_\perp^{-1}$, from which the pressure values can be derived.



(a) Profile



(b) Texture

Figure 2.4: A profile along u coordinate and a 25×5 texture for an arbitrary geometry. Texture and profile values show the derived pressure value from the sum of reciprocals of orthogonal speed components. Pressure values are dropping from the left (inlet) to the right (outlet). The x-axis in the pressure profile shows the relative position along the u-coordinate in percent and the y-axis shows the pressure value.

Textures Textures divide a facet in two directions. Orientated along a facet's local 2d coordinates u, v with $u, v \in [0, 1]$, a grid is created with $N \times M$ uniform cells. When a particle collides with a textured facet, its collision point is mapped to the texture coordinates $t_u \in [0, N-1], t_v \in [0, M-1]$, where $t_u, t_v \in \mathbb{N}^+$. Given a particle hit location $\mathbf{p} = (p_u, p_v)$ on facet f, where p_u and p_v are the coordinates along the local u, v coordinates of f, we can calculate the position

$$t_u = \lfloor p_u \times N \rfloor\,, t_v = \lfloor p_v \times M \rfloor\,, \tag{2.36}$$

where we require $t_u \in [0, N]$ and $t_v \in [0, M]$.

Each cell in the texture grid collects data about the particle hits it receives. This data can include the number of hits, further distinguished in the number of desorptions, absorptions or reflections, as well as the sum of orthogonal momentum change $\Sigma I_\perp$ and the sum of reciprocals of orthogonal speed components $v_\perp^{-1}$. An example showing a texture on an arbitrary Molflow geometry is shown in sketch 2.4b. Here, a 25×5 texture is deployed to count the sum of reciprocals of orthogonal speed components $v_\perp^{-1}$. Each texture cell represents a derived pressure value.

Trace processing: Next iteration

Depending on the type of facet with which the particle collided, it will advance differently. In case a particle bounces off a facet, the particle state tuple will be set as follows. The reflection point is the new origin, the direction is calculated according to Knudsen's cosine law (see equation (2.11)) or from specular reflection. The velocity of a particle depends on the energy and momentum exchange with a wall, which we can describe with the thermal accommodation coefficient A_{acc}. Total thermalisation is given by $A_{acc} = 1$, resulting in a new velocity value v_{wall} that depends only on the modified speed distribution (see equation (2.24)), where the facet temperature is taken into account. The new particle velocity is then given by:

$$v_{new}^2 = v_{old}^2 + A_{acc}(v_{wall}^2 - v_{old}^2) \,, \tag{2.37}$$

with the incident velocity v_{old}. Given the new particle speed and direction, the counters $\Sigma I_\perp$ and $v_\perp^{-1}$ have to be incremented again in consideration of the outgoing particle's momentum and reciprocal speed.

Post processing: Physical quantities

Based on the statistical quantities, the following physical quantities are derived, which are of interest for vacuum systems.

Impingement rate z is the rate of number of particle collisions $N_{hit,real}$ with a facet per second and unit area:

$$z = \frac{dN_{hit,real}}{dt \cdot A} = \frac{N_{hit} \cdot K_{\frac{real}{test}}}{A} \,. \tag{2.38}$$

Here $K_{\frac{real}{test}}$ is the number of molecules per second that is represented by a test particle, given by equation (2.6), and A is the area of the corresponding facet. N_{hit} represents the number of test particles hits.

Pressure p can be calculated by

$$p = \frac{F}{A} = \frac{\sum dI}{dt \cdot A} = \frac{\sum I_\perp \cdot K_{\frac{real}{test}}}{A} \,. \tag{2.39}$$

Where the rate of momentum change of the inbound and outbound particles $\sum dI/dt$ is approximated by the product of the statistical counter $\sum I_\perp$ and the factor $K_{\frac{real}{test}}$.

Density $< n >_{volume}$ defines the average particle density in a given volume adjacent to a facet. The particle density is derived from the relationship between the rate of particle impingement on a surface and their velocities upon collision (see details in Marton Ady (2016)). It is calculated with:

$$< n >_{volume} = z_{surface} \cdot < v_\perp >_{coll} = \frac{\Sigma \frac{1}{v_\perp} \cdot K_{\frac{real}{test}}}{A} \,, \tag{2.40}$$

where $z_{surface}$ is the impingement rate on a surface, measuring how frequently particles collide with the surface per unit area per second, and $\langle v_\perp \rangle_{coll}$ is the average perpendicular velocity at collision.

Summary

In this section, we summarize the key steps in the Molflow's Monte Carlo simulation process as implemented. These steps describe a single iteration for one particle, which was sketched in figure 2.2.

1. **Calculation of Random Origin:** Selecting a starting point for particles in the simulation by randomly choosing a facet and position.
2. **Calculation of Random Direction:** Assigning a random trajectory to each particle, simulating natural variability in their angles.
3. **Calculation of Random Speed Value:** Determining the speed of particles using a method that ensures realistic distribution based on physical principles.
4. **Ray Tracing to Find Collision Point:** Using ray tracing algorithms to track the path of each particle until it collides with a surface.
5. **Incrementation of Hit Counters:** Updating statistical counters (local facet counters and global simulation counters) to record the interactions of particles with different facets.
6. **Setup of the Next Iteration:** Preparing for the next iteration of the simulation, accounting for different possible outcomes of particle interactions. For example, on reflection a new particle direction and velocity is calculated. Hit counters are updated to reflect momentum change with the outgoing direction.

2.2.4 Time-dependent simulation

Molflow offers two distinct types of simulations: steady-state and time-dependent. While steady-state simulations are defined to be in equilibrium, providing a single static result, time-dependent simulations are dynamic. They allow analysis of vacuum systems at multiple time points, accommodating changes in the vacuum system's parameters over time. This adds a layer of complexity, as statistics need to be gathered not only spatially for each facet, but also temporally. System parameters, such as outgassing rates or sticking factors may vary over time. In such systems it is of interest to track particle events at the specific time at which the hit occurs.

Following the original implementation by Marton Ady (2016), time-dependent parameters like a facet's outgassing rate or sticking factor can be set to vary over time. This is achieved by specifying a series of time-value points, where the parameter value at a given time is determined by linear interpolation between these points or by constant extrapolation outside this range. Time moments are central to time-dependent simulations in Molflow. They are defined as intervals corresponding to specific points in time, allowing for the accumulation of statistical data over these periods. Each time point t_i in a series of time moments is associated with a time bin covering the range $[t_i - t_w/2, t_i + t_w/2)$, where t_w is the time window size. This approach ensures that statistics are gathered over a sufficiently large sample size to be meaningful. While the calculation of the steady-state quantities is straightforward, time-dependent quantities have to be calculated for every i-th moment. This allows for a detailed analysis of the system's behavior over time, accommodating dynamic changes in parameters and providing deeper insights into the system's performance.

For steady-state simulations in Molflow, we can derive the physical quantities as elaborated in chapter 2.2.3. Using the hit counters for the i-th moment, we can derive the corresponding time-dependent quantities for the impingement rate, the pressure and the density. They are noted noted with an index corresponding to the i-th moment. The impingement rate for the i-th moment, z_i, can be calculated as follows:

$$z_i = \frac{N_{hit,i} \cdot K_{\frac{real}{test}}}{t_{window} \cdot A}, \tag{2.41}$$

where $N_{hit,i}$ represents the number of test particle hits during the i-th moment. By dividing by the time window t_{window} to obtain an average rate over this interval. The pressure at a specific time moment, p_i, can be calculated using the following equation:

$$p_i = \frac{\Sigma I_{\perp,i} \cdot K_{\frac{real}{test}}}{t_{window} \cdot A}. \tag{2.42}$$

Here, $\Sigma I_{\perp,i}$ represents the sum of the perpendicular components of the momentum change of particles impacting the surface at the i-th moment. Similarly, the particle density at the i-th moment, $\langle n \rangle_i$, is given by:

$$\langle n \rangle_i = \frac{\Sigma \frac{1}{v_{\perp,i}} K_{\frac{real}{test}}}{t_{window} \cdot A}\,, \tag{2.43}$$

where $\Sigma \frac{1}{v_{\perp,i}}$ sums the inverse of the perpendicular velocities of particles impacting the surface during the i-th moment.

Time-dependent simulations and enhancements to the underlying algorithms and data structures are elaborated in more detail in chapter 5.3.

2.2.5 Pseudo Random Number Generator

Since many parameters in Molflow's simulation algorithm are dependent on uniformly distributed variables, it is important to use a Pseudo Random Number Generator (PRNG) with good statistical properties. In Molflow, random numbers are repetitively used for particle tracing. They are used to calculate the particle origin, the particle trajectory and the particle velocity (see chapter 2.2.3). Further, they are used to model probabilities e.g. for transparent facets (see chapter 2.2.3). A PRNG is an algorithm that generates sequences of random numbers that reflect the properties of real random numbers as accurately as possible. Typically, the algorithm starts from an initial value, the seed, and calculates a pseudo-random number based on its advanced state. As PRNGs are deterministic algorithms, they return the same sequence of random numbers given the same initial value. The qualities of a PRNG are related e.g. to the way the generated numbers in a sequence are distributed or to its period, which is the length of a sequence, after which the same sequence will be generated again. Consider the Monte Carlo simulation to estimate π. The simulation randomly places points within a square bounding the unit circle and counts how many points fall inside the circle versus outside. The ratio of the points inside the circle to the total number of points approximates the ratio of the circle's area to the square's area. The area of the circle is πr^2 and the area of the square is $4r^2$, where r is the radius of the circle. The ratio of both is $\pi/4$. Now, if the PRNG used in this simulation has a short period or exhibits patterns, certain areas of the square may be sampled more frequently than others. Such a non-uniform sampling leads to an inaccurate estimation of the circle's area. For instance, if the PRNG repeatedly generates numbers clustering towards one corner of the square, the simulation will overestimate the area of that quadrant of the circle, skewing the overall result. A well-known example for a bad PRNG is RANDU, a LCG (Linear congruential generator) algorithm. LCG algorithms repetitively apply linear transformations to generate a series of pseudo random numbers. RANDU is defined by the recurrence using the linear congruential formula:

$$\mathrm{R}_{i+1} = (65539 \cdot \mathrm{R}_i) \mod 2^{31}\,, \tag{2.44}$$

where R_{i+1} is the $(i+1)$-th generated random number, and R_i is the previous random number, where $i = 0$ would be the seed. George Marsaglia (1968) proved, that by generating n-dimensional points using RANDU, these points would fall in a small amount of parallel hyperplanes. Hyperplanes represent flat, $n-1$-dimensional surfaces in an n-dimensional space, which they separate in two distinct regions. For $n = 3$ it is easy to show, that only 15 hyperplanes are sampled, as seen in figure 2.5. Here, every three consecutive random numbers form a point $\mathbf{p} = \mathrm{R}_{i+2}, \mathrm{R}_{i+1}, \mathrm{R}_i$. This proves bad statistical qualities of the generator and easily leads into bias. In

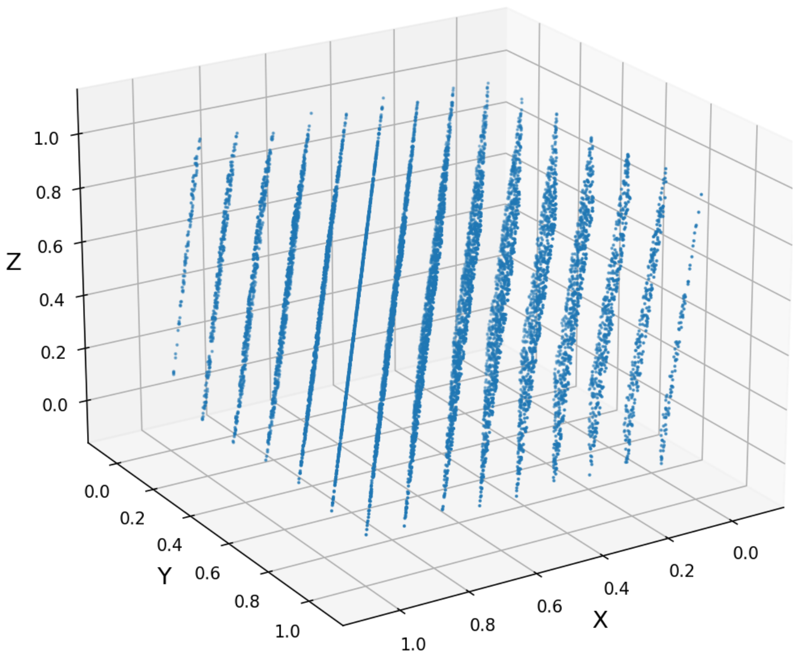


Figure 2.5: 3D visualization of $N = 10000$ points, each represented by three consecutive random numbers generated with the RANDU generator. All points clearly fall into 15 hyperplanes.

many Monte Carlo applications, such as Molflow, the Mersenne Twister algorithm is deployed, due to its long period, but nowadays it is viewed as lacking in several aspects as it fails some of the common statistical quality tests and is comparably slow (see O'Neill, 2014). While the statistical quality seems to be sufficient for Monte Carlo simulations, the feasibility of a faster algorithm should be considered. An initial profiling of Molflow's simulation code suggests that random number generation may account for around 2% of simulation time (see chapter 3.6) in simple scenarios. Leaving only room for marginal improvements. In hindsight of the development of a new simulation kernel for GPUs, other generators are taken into account.

Various PRNGs have been compared by O'Neill (2014) and a new algorithm, the

Permuted Congruential Generator (PCG) family, has been proposed. XorShift* and generators based on the PCG family are some of the more interesting choices due to their attributed performance and good statistical properties. PCG algorithms are combinations of LCG with permutation functions. Xorwow (see G. Marsaglia (2003)) is the default PRNG for NVIDIA's RNG library cuRAND[4]. Xorshift* and Xorwow are LCG algorithms. L'Ecuyer (2012) gives some general insight into PRNG and into applying PRNG in parallel computing environments. For the implementation of Molflow's GPU kernel, we decided to use cuRAND's Xorwow implementation due to its ease of use and proven reliability in many applications. Our GPU implementation is discussed in chapter 8.

2.2.6 State of the art

There are various commercial and non-commercial solutions for simulating ultra-high vacuum problems. The Molecular Flow Module of the COMSOL Multiphysics® Software[5] utilises an algorithm based on the Angular Coefficient (AC) method (see chapter 2.2.1) to simulate steady-state free molecular flows in UHV conditions. The AC method divides a geometry into N surface elements and calculates a visibility factor for each element with the others, leading to a $N \times N$ linear system that needs to be solved. Compared to Molflow's ray tracing simulations, COMSOL's angular coefficient is usually slower and further does not support time-dependent simulations. In ANSYS[6], one can not directly simulate vacuum problems. Using analogies between thermal and vacuum calculations, a heat flow simulation can be used to acquire solutions for a vacuum model.

2.2.7 Case studies

Molflow has become a quasi-standard application for ultra high vacuum simulations and is not only used by engineers and physicists in the accelerator community, but also in other domains. At CERN it has been used for many components related to the LHC. Veness et al. (2019) developed a beam gas curtain monitor for the high luminosity upgrade of the LHC. Molflow simulations were used to suggest several optimizations to the whole system. A test setup has been installed in the LHC to verify the results.

Molflow has been used in the domain of space exploration by Alred et al. (2021) as part of a contamination study for NASA's SPHEREx mission. SPHEREx is a low-orbit space observatory. Here, water outgassing scenarios have been simulated

[4]Accessed on 29/11/2023: https://developer.nvidia.com/curand
[5]Accessed on17/12/2022: `https://www.comsol.com/molecular-flow-module`
[6]Accessed on17/12/2022: `https://www.ansys.com`

as part of a model to predict water contamination risks and to design decontamination solutions. At certain temperatures H_2O is outgassed from the walls and then potentially accumulated on top of the telescope's optics, leading to disturbed results. In figure 2.6 we show the Molflow model of the SPHEREx telescope.

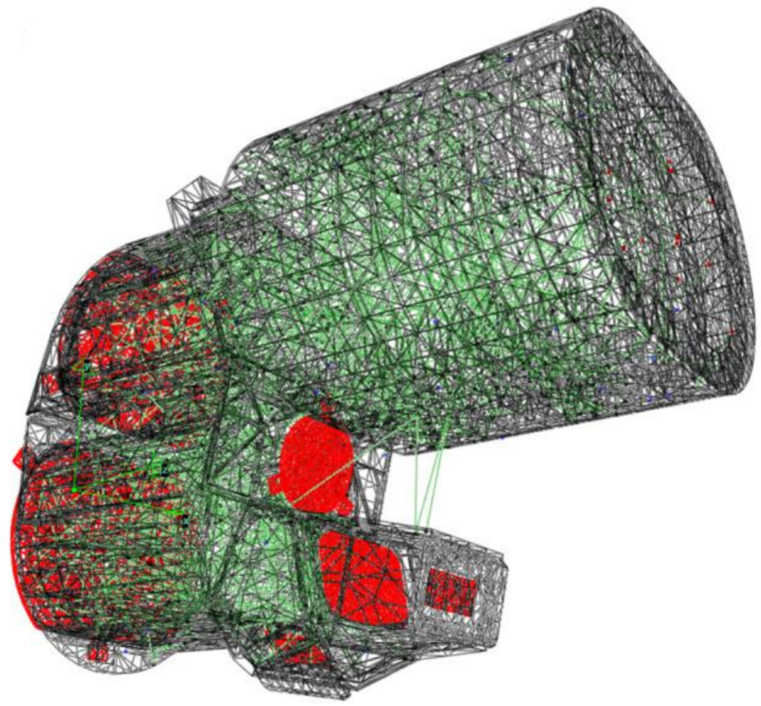


Figure 2.6: Molflow model of the SPHEREx telescope. Green lines represent the water molecule transport and red patches represent optical surfaces of the system (Alred et al., 2021).

2.2.8 Synrad

While the work of this thesis was mainly focussed on Molflow, most contributions are directly related to both simulation tools. Molflow and Synrad are sharing the same computational back-end and large parts of the graphical front-end allowing for preprocessing and postprocessing steps. The fundamental difference lies in the Monte Carlo model, which we want to briefly elaborate on. For a deeper understanding, we direct users to the work of Marton Ady (2016) and Roberto Kersevan and Marton Ady (2019).

Synrad is a tool for approximating flux and power distributions with a Monte Carlo model for Synchrotron Radiation (SR). Synchrotron radiation is emitted by charged particles, such as electrons, when they are accelerated radially. This is in particular the case for circular accelerators, such as the LHC at CERN. In particle accelerators, this occurs when the particles move at near-light speeds and are forced into a curved path by magnetic fields. The rapid change in direction causes the particles to emit energy in the form of synchrotron radiation. The synchrotron radiation itself is composed of photons. These photons carry the energy emitted by the accelerating electrons.

In Synrad's Monte Carlo model, photons originate from a user-defined beam calculated by a set of magnetic regions and beam parameters. This allows to calculate their origin based on a random distribution and a photon's trajectory. Photons are then emitted from this beam, their path is traced until a hit with a surface, leading to a reflection and absorption, where backscattering and transmission probabilities can be calculated, optionally (Roberto Kersevan and Marton Ady, 2019).

When this photon-rich synchrotron radiation interacts with surfaces and different materials, estimated in Synrad with the absorbed photon flux, it can result in various phenomena such as photon stimulated desorption.

2.2.9 Ultra high vacuum and Synchrotron radiation problems

Photon stimulated desorption (PSD) is a common phenomenon that has to be considered for vacuum components that are exposed to synchrotron radiation. In a particle accelerator, incident SR photons may lead to the desorption of gas molecules from the material of the accelerator surface. As this phenomenon may contribute significantly to the gas load in such systems, degrading the efficiency and performance of particle accelerators, it is important to consider and minimise this effect when designing vacuum components for this problem domain. In Synrad and Molflow, this effect can be simulated by first generating a flux absorption with a Synrad simulation, which is then imported as an outgassing distribution in Molflow. Marton Ady (2016) researched in great detail how these effects can be simulated with both Synrad and Molflow.

For the remainder of the thesis, our attention is primarily on Molflow. It's important to note that many of the insights and contributions presented here are equally applicable to both Molflow and Synrad, thanks to their shared software foundation. This is discussed in greater detail in chapter 3. Additionally, a summary of the specific contributions made to Synrad as part of this work is provided in the concluding chapter 9.

2.3 Ray tracing

Ray tracing (RT) is a computational technique for simulating the path of light through a scene. It is a fundamental concept in both computer graphics and for many

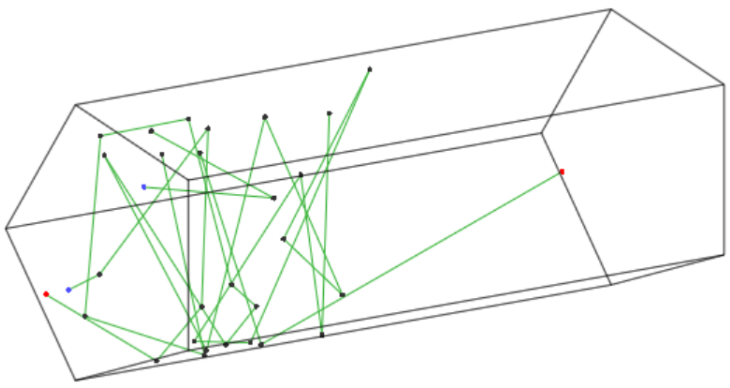


Figure 2.7: Ray tracing problem sketched in Molflow. Here a particles enter from the facet at the left side (a desorption facet), following the particle trajectory (green line) the algorithm tries to identify the facet which the particle collides first with. Particles can exit this geometry either at facet on the left or the right side (absorption facets). The dots connecting the trajectory lines represent desorptions (blue dot), reflections (black dot) or absorptions (red dot).

scientific simulations. In computer graphics, ray tracing is employed to create highly realistic images by accurately rendering effects such as shadows, reflections, and refractions. This realism is achieved by tracing the path of light rays as they interact with various surfaces in a virtual environment. Beyond its typical applications in creating visual content, ray tracing can be applied for physical simulations such as Molflow. Molflow utilises ray tracing methodologies to simulate molecular flows in vacuum environments. Instead of tracing the paths of light rays, in Molflow tracing the trajectories of particles is of interest. In Molflow a ray is depicted as a particle, where the collision point along the particle's trajectory with a facet is of interest. A facet is an arbitrary, non-self-intersecting polygon described by its vertices. An example for simple a model in Molflow is sketched in figure 2.7.

This chapter aims to provide a comprehensive overview of ray tracing, covering its principles and its implementation. We will begin by examining the basic concepts underlying ray tracing as explained in great detail by Akenine-Möller, Haines, and Hoffman (2008) and Pharr, Jakob, and Humphreys (2017). This is followed by a brief discussion about optimizing techniques leading to the introduction of acceleration structures to optimize the ray tracing computations.

2.3.1 Basics

Ray tracing can be best understood by examining its typical pipeline, which consists of three main stages: ray generation, ray tracing, and post-processing (see figure 2.2). In this thesis we use some terms interchangeably. For simplicity, we use the term *ray*, where in the case of Molflow a particle's geometric representation is meant. *Polygons* refer to the geometric descriptions of facets.

Given a geometry $\mathcal{G}$ that defines a polygon scene represented by the set of polygons Ω, containing $N \in \mathbb{N}$ polygons, and a ray r, an orientated line, we can define a ray tracing query $\textsc{rt}(\Omega, r)$. The result of a ray tracing query is typically the spatially first intersected polygon $P_i \in \Omega$, $i \in [1, N]$ or no polygon. The ray generation to retrieve the origin and the direction, as well as the post-processing steps have been discussed for Molflow's Monte Carlo model in chapter 2.2.3. Here, we focus on the algorithm that returns an intersection for the query $\textsc{rt}(\Omega, r)$.

Query types

Ray tracing describes a line-object intersection problem, where most commonly the polygon that is intersected by the line first is of interest. For example, in a scene with multiple polygons, the Closest Hit kernel determines which polygon a ray intersects first for the query $\textsc{rt}(\Omega, r)$. In addition to the Closest Hit kernel, a ray tracer can deploy different traversal algorithms depending on the geometric query (see Gribble, Naveros, and Kerzner, 2014). The Any Hit kernel simply returns whether a ray intersects any of the polygons inside the geometry and returns the first polygon that has been identified to be intersected. When a query is made following the Multi Hit traversal, information is returned about the N_k closest intersected primitives. Closest Hit and All Hit traversal kernels are specialisations of Multi Hit traversal, where $N_k = 1$ respectively $N_k = \infty$. When a ray intersects a polygon, we calculate the distance from the ray's starting point to the intersection point. This distance, along with the polygon's ID and the intersection's coordinates on the polygon, is recorded as a tuple, providing detailed information about the intersection. This information is formed by the tuple $(P_{ID}, t_{hit}, u_{hit}, v_{hit})$. Here, P_{ID} is the ID of the polygon, t_{hit} is the calculated distance, and u_{hit} and v_{hit} defined the local 2D coordinates of the intersected polygon. The closest hit can be labelled as $\textsc{hit}$, where the ray is colliding with the closest polygon and an interaction event occurs, e.g. a reflection (see chapter 2.2.3). In case no interaction is identified, this is tracked as a $\textsc{miss}$. Here, the ray is leaving the system and, in Molflow, has to be terminated.

Transparent facets: Molflow deploys a special type of Closest Hit kernel, where the closest primitive is of main interest, but intersected transparent facets are gath-

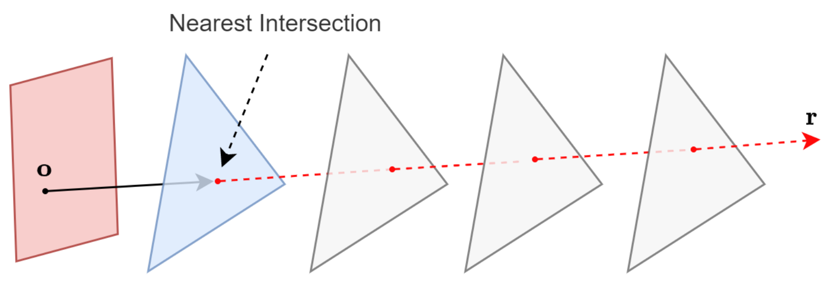


Figure 2.8: A ray is following trajectory **r** from origin **o** on the red facet. Using the Closest Hit kernel, the ray tracing query only returns the intersected triangle that is the closest (blue facet). The other intersected triangles, following the red trajectory line, are not returned (white facets).

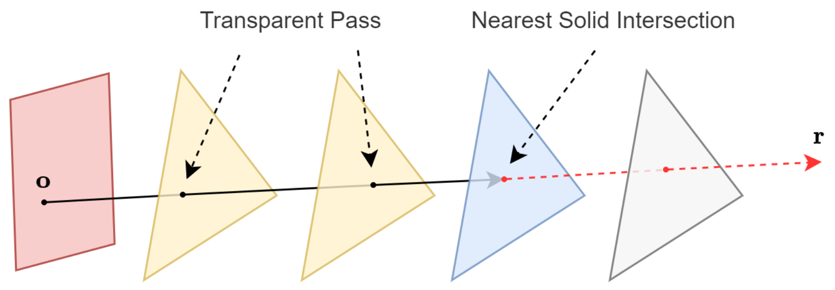


Figure 2.9: A ray is following trajectory **r** from origin **o** on the red facet. Using the Any Hit kernel, all intersections are evaluated. Here, the ray tracing query returns the transparent triangles (yellow facets) and the closest solid intersection (blue facet). The other intersected triangle, following the red trajectory line, is not returned (white facet).

ered along the path. Transparent facets (as described in chapter 2.2.3) do not influence the particle directly, but they serve the purpose of gathering statistics. The opacity of a transparent is given by the opacity value $\tau \in [0,1]$. An opacity value of $\tau = 0$ denotes a solid facet, while an opacity value of $\tau = 1$ denotes a perfectly transparent facet. Values in between have to be evaluated ad hoc with a random number. Given a randomly generated number $\mathrm{R} \in [0,1]$, when a geometric intersection is found with a transparent facet, we can determine directly from the opacity value, whether a particle rebounds or is passed through. A facet acts as a transparent facet, where a particle passes through if $\mathrm{R} \leq \tau$. A passed transparent facet and its hit location are collected in an array, only if $t_{trans} < t_{closest}$. Here, t_{trans} is the intersection distance between the ray origin and the transparent facet. $t_{closest}$ is the intersection distance with the facet, that the ray collides with, the closest hit with a solid facet. A collection of passed transparent facets is then evaluated in the trace processing step.

2.3.2 Coordinate system

Efficient ray tracing algorithms may require certain coordinate systems or transformations. In ray tracing, transforming coordinates from a 3D global space to a local 2D space is a common technique, typically referred to as UV mapping. Particularly, it can be used for texture mapping and simplified geometrical operations. Geometries are commonly expressed as global coordinates in 3D space. Global coordinates relate all objects to each other. They are represented by the tuple $(x, y, z) \in \mathbf{R}^3$. Local coordinates refer to normalized coordinates in 2D space on a surface. They are represented by $(u, v) \in [0, 1]^2$ coordinates.

For the transformation we utilize three vectors. The surface normal $\mathbf{N} \in \mathbf{R}^3$, which is perpendicular to the surface, defines the orientation of the surface. Next, defining a tangent $\mathbf{T} \in \mathbf{R}^3$ and a bitangent $\mathbf{B} \in \mathbf{R}^3$, where both are orthogonal to $\mathbf{N}$, we can form a basis for the surface's local coordinate system. These vectors should be normalized so that $|\mathbf{N}| = 1$, $|\mathbf{T}| = 1$, $|\mathbf{B}| = 1$. In our case, the tangent $\mathbf{T}$ is typically chosen to align with an arbitrary edge of the surface. Then, we can compute the bitangent $\mathbf{B}$ with:

$$\mathbf{B} = \mathbf{N} \times \mathbf{T}\,, \tag{2.45}$$

so that $\mathbf{T}$, $\mathbf{B}$, and $\mathbf{N}$ are orthogonal to each other. After computing $\mathbf{B}$, it should also be normalized to ensure numerically that it is a unit vector. Together, $\mathbf{T}$, $\mathbf{B}$, and $\mathbf{N}$ form an orthonormal basis for the surface's local coordinate system.

We construct a transformation matrix to map global coordinates to the local coordinate system. Using $\mathbf{T}$, $\mathbf{B}$, and $\mathbf{N}$ as columns we get:

$$\mathbf{M} = \begin{bmatrix} T_x & B_x & N_x \\ T_y & B_y & N_y \\ T_z & B_z & N_z \end{bmatrix}\,, \tag{2.46}$$

where the indices represent the corresponding global coordinate axis. We can then transform a point $\mathbf{p}$ to local coordinates:

$$\mathbf{p}_{local} = \mathbf{M}^{-1} \times \mathbf{p}\,. \tag{2.47}$$

The resulting $\mathbf{p}_{local}$ will have its u and v components corresponding to the coordinates in the surface's 2D space. After obtaining the local coordinates $\mathbf{p}_{local}$ for a point $\mathbf{p}$, it is essential to ensure that these coordinates fall within the range $[0, 1]^2$ to maintain consistency and compatibility with common practices in UV mapping.

This can be achieved by calculating the minimum and maximum values of the u and v coordinates across all vertices of the surface, forming a 2D bounding box. Then, the u and v coordinates of each point can be normalized using the minimum and maximum values of the bounding box.

In practice, it's often more practical to shift each point towards a reference point, like an arbitrary polygon vertex $\mathbf{p}_0$, and directly compute u and v coordinates:

$$\begin{aligned} u &= \mathbf{T} \cdot (\mathbf{p} - \mathbf{p}_0)\,, \\ v &= \mathbf{B} \cdot (\mathbf{p} - \mathbf{p}_0)\,. \end{aligned} \tag{2.48}$$

By normalizing these coordinates using the bounding box's minimum and maximum values:

$$\begin{aligned} \bar{u} &= \frac{u - u_{\min}}{u_{\max} - u_{\min}}\,, \\ \bar{v} &= \frac{v - v_{\min}}{v_{\max} - v_{\min}}\,, \end{aligned} \tag{2.49}$$

where $u_{\min}$ and $u_{\max}$ represent the minimum and maximum u coordinates across all vertices, and $v_{\min}$ and $v_{\max}$ denote the minimum and maximum v coordinates. This guarantees numerically that each 2d coordinate $(\bar{u}, \bar{v})$ falls within the range $[0, 1]^2$.

Figure 2.10 illustrates the mapping of 3D coordinates to 2D coordinates. In ray tracing, these transformations are essential for texture mapping and the calculation of effects following intersections with surfaces. Further, they can be used for simplifying ray-surface intersection calculations.

2.3.3 Ray tracing algorithm

Ray tracing fundamentally revolves around the concept of intersections - determining where and how rays interact with objects in a scene. These intersections are crucial for simulating realistic behaviors of light or particles. To build a solid understanding, we begin by explaining the simple intersection algorithms and elaborate further details for a more optimized solution.

Naively, to find the closest point of collision for a ray and a given geometry $\mathcal{G}$ with all of its polygons, intersections with all N polygons of the geometry have to be evaluated. We give a naive implementation for the intersection algorithm in algorithm 2. For simplification, it uses the inverse of the ray direction $\mathbf{r}_i = \{\frac{1}{r_x}, \frac{1}{r_y}, \frac{1}{r_z}\}$ (see line 2). The intersection tests involve divisions with the direction vector components. Using the inverse simplifies the calculations. Divisions by zero can be avoided early on. Furthermore, it is computationally cheaper to use multiplications instead of divisions. The algorithm is naive, because it has to test against all facets for intersection (see line 4). Hence, it is $O(N)$, where N is the amount of facets. We present optimized solutions $O(\log N)$ in chapter 2.4.

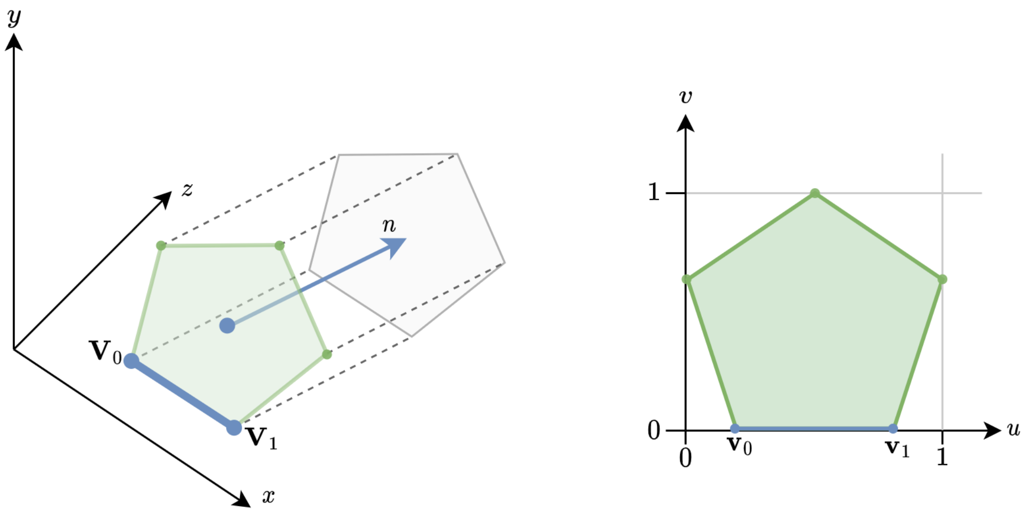


Figure 2.10: The 3D world coordinates (x, y, z) for the front face (green) of a prism are transformed to local 2D coordinates $(u, v) \in [0, 1]^2$ using the facet normal $\mathbf{N}$ and the tangent along $(\mathbf{V}_0, \mathbf{V}_1)$. Here, coordinates $\mathbf{V}_0, \mathbf{V}_1$ are mapped to $\mathbf{v}_0, \mathbf{v}_1$, respectively.

Algorithm 2 Naive Ray-polygon intersection algorithm

```
1: function INTERSECT(Particle p, Direction r, Geometry $\mathcal{G}$)
2:     $r \leftarrow$ INVERSERAYDIRECTION($r$)
3:     $t \leftarrow \infty$                                  ▷ Used to store the shortest distance
4:     for facet in FacetList of $\mathcal{G}$ do                  ▷ Naive check for all facets
5:         INTERSECTPOLYGON(p,r,facet)
6:         $t_f \leftarrow$ INTERSECTIONDISTANCE
7:         if Intersection found $\wedge t_f < t$ then
8:             $closest\_facet \leftarrow facet$
9:             $t \leftarrow t_f$
10:        end if
11:    end for
12: end function
13:
14: function INTERSECTPOLYGON(Particle p, Direction r, Polygon f)
15:     if LINE-PLANE INTERSECTION $\wedge$ POINT-IN-POLYGON CHECK then
16:         return true
17:     end if
18:     return false
19: end function
```

A ray-polygon intersection in 3D coordinates can be computationally expensive. In many cases, it is possible to reject an intersection test with a much simpler test. In our implementation `IntersectPolygon` (see line 14 of algorithm 2), first, we test whether the ray intersects the 3D-plane of the polygon. In case of an intersection, a conclusive test can be conducted with a point-in-polygon tests in 2D coordinates.

In the following sections, we elaborate the details of the ray-polygon-intersection algorithm then we explain how the $O(N)$ algorithm can be optimised using so-called acceleration data structures.

2.3.4 Line-plane intersection

The ray tracing problem in Molflow is described in parametric form. A ray is represented with:

$$\mathbf{I} = \mathbf{o}_{ray} + \mathbf{r_{dir}}t, \quad t \in \mathbb{R}\,, \tag{2.50}$$

where $\mathbf{I}$ are all valid points on the ray, $\mathbf{o}_{ray}$ is its origin and $\mathbf{r_{dir}}$ is the ray's normalized direction and t is some scalar for the direction that depicts the distance from the origin. Because intersection tests with arbitrary polygons in 3 dimensions are computationally expensive, the algorithm starts by utilising some simplifications. For each polygon, we can define the plane spanned by the $\mathbf{u}$ and $\mathbf{v}$ coordinate vectors:

$$\mathbf{p} = \mathbf{o}_{fac} + \mathbf{u}u + \mathbf{v}v, \quad u, v \in \mathbb{R}\,, \tag{2.51}$$

where $\mathbf{p}$ are all valid points on the plane, $\mathbf{o}_{fac}$ is its origin and u, v are some scalars. The ray collides with the plane when there is an equal point of the ray and the plane; thus, equating (2.50) and (2.51) we get the equation:

$$\mathbf{o}_{ray} + \mathbf{r_{dir}}t = \mathbf{o}_{fac} + \mathbf{u}u + \mathbf{v}v\,, \tag{2.52}$$

which we can rewrite as

$$\mathbf{o}_{ray} - \mathbf{o}_{fac} = \mathbf{u}u + \mathbf{v}v - \mathbf{r_{dir}}t\,, \tag{2.53}$$

and in matrix form as

$$\begin{bmatrix}\mathbf{o}_{ray} - \mathbf{o}_{fac}\end{bmatrix} = \begin{bmatrix}\mathbf{u} & \mathbf{v} & -\mathbf{r_{dir}}\end{bmatrix} \begin{bmatrix} u \\ v \\ t \end{bmatrix}\,. \tag{2.54}$$

We can solve the corresponding system of linear equations for u, v and t, which is commonly achieved via Cramer's rule (see Shirley and Marschner, 2009). Given the system of linear equations in multiplication form

$$\mathbf{b} = \mathbf{A}\mathbf{x}\,, \tag{2.55}$$

where $\mathbf{x} = (u, v, t)^T$ is our vector of variables. Then we can express the individual solutions x_i with

$$x_i = \frac{\det(\mathbf{A}_i)}{\det(\mathbf{A})} = \frac{|\mathbf{A}_i|}{|\mathbf{A}|}\,, \tag{2.56}$$

but only if the determinant of $\mathbf{A}$ is not zero. $\mathbf{A}_i$ denotes the matrix, where we use the vector $\mathbf{b}$ as the i-th column. That means for

$$\mathbf{A} = \begin{bmatrix} x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \\ x_3 & y_3 & z_3 \end{bmatrix} \tag{2.57}$$

the solutions can be calculated as:

$$u = \frac{1}{|\mathbf{A}|}\begin{vmatrix} b_1 & y_1 & z_1 \\ b_2 & y_2 & z_2 \\ b_3 & y_3 & z_3 \end{vmatrix}\,, \tag{2.58}$$

$$v = \frac{1}{|\mathbf{A}|}\begin{vmatrix} x_1 & b_1 & z_1 \\ x_2 & b_2 & z_2 \\ x_3 & b_3 & z_3 \end{vmatrix}\,, \tag{2.59}$$

$$t = \frac{1}{|\mathbf{A}|}\begin{vmatrix} x_1 & y_1 & b_1 \\ x_2 & y_2 & b_2 \\ x_3 & y_3 & b_3 \end{vmatrix}\,. \tag{2.60}$$

For system (2.54), where we use the scalar triple product to describe each determinant, we get the following results:

$$u = \frac{(\mathbf{v} \times -\mathbf{r_{dir}}) \cdot (\mathbf{o}_{ray} - \mathbf{o}_{fac})}{-\mathbf{r_{dir}} \cdot (\mathbf{u} \times \mathbf{v})}\,, \tag{2.61}$$

$$v = \frac{(-\mathbf{r_{dir}} \times \mathbf{u}) \cdot (\mathbf{o}_{ray} - \mathbf{o}_{fac})}{-\mathbf{r_{dir}} \cdot (\mathbf{u} \times \mathbf{v})}\,, \tag{2.62}$$

$$t = \frac{(\mathbf{u} \times \mathbf{v}) \cdot (\mathbf{o}_{ray} - \mathbf{o}_{fac})}{-\mathbf{r_{dir}} \cdot (\mathbf{u} \times \mathbf{v})}\,. \tag{2.63}$$

To optimise the algorithm, it makes sense to precalculate $\mathbf{w} = \mathbf{u} \times \mathbf{v}$ as it is just another constant for the facet. Explicitly precalculating certain constants removes redundancy, which guarantees computational efficiency. For every intersection we can reuse values for the determinant $\det(\mathbf{A}) = -\mathbf{r_{dir}} \cdot (\mathbf{w})$ and the difference between the ray's and the facet's origin $\mathbf{o}_{fr} = \mathbf{o}_{ray} - \mathbf{o}_{fac}$.

Given our ray tracing problem, a prerequisite for an intersection is:

$$\det(\mathbf{A}) \neq 0\,, \tag{2.64}$$

which ensures that ray and the facet's plane are not coplanar, otherwise there could not be an intersection. For 1-sided facets, a check for

$$\det(\mathbf{A}) > 0 \tag{2.65}$$

guarantees only intersections with the front side of the facet. Additionally the following constraints need to be satisfied:

$$u, v \in \; [0,1]\,, \qquad \text{(so that it is inside the bounding rectangle)} \tag{2.66}$$

$$t > \; 0\,. \qquad \text{(so that it is in the right direction)} \tag{2.67}$$

The point (u, v) is then used to check if the ray really hit the facet and not only its rectangular bounding box. A bounding box surrounds a given facet completely. Here, a bounding box is the minimum axis aligned rectangle that bounds the facet's coordinates. Only when the bounding box is intersected, a computationally more expensive point-in-polygon algorithm is called, which is explained in the following chapter 2.3.5. Additionally, transparent facets have to be handled in more detail. An implementation for `HandleTransparentHit` (see line 12) has to consider the following. For a partially transparent facet f with the opacity value $\tau_f \in [0,1]$, the hit has to be registered only as a real hit if r $> \tau_f$, where r is a pseudo random number.[7] A slightly simplified version of the algorithm for `Line-plane intersection` using Cramer's rule is shown in algorithm 3.

2.3.5 Point in polygon

For Molflow, we use a modified version of the Winding number algorithm (Sunday, 2021) that works with both convex and concave polygons and is independent in regards to the orientation of the polygon's normal. A good analysis of common point-in-polygon algorithms has been carried out by Schirra (2008). The Winding number algorithm is computationally efficient and has shown good results for Molflow. The algorithms works by counting the number of times the polygon winds around a given point $\mathbf{p} = (u, v)$. An efficient version gives the winding number by utilizing a positive ray in **v**-direction starting from **p**. An edge crossing the ray above adds and otherwise subtracts from the winding number.

Using local 2D coordinates $\mathbf{p}_i = (u_i, v_i)$ to describe the polygon, for every edge of the polygon $e = (\mathbf{p}_1, \mathbf{p}_2)$ we check the following conditions:

- Check if **u**-value of **p** is within $[u_{p_1}, u_{p_2}]$.
- Check if **v**-value of **p** is above or below the line connecting the points v_{p_1} and v_{p_2}.

[7]So, for $\tau_f = 1$ a hit always reflects. For $\tau_f = 0$, a hit is always considered transparent and is only of statistical nature.

Algorithm 3 Line-Plane intersection via Cramer's rule

```
function LinePlaneIntersection(Ray R, Facet f, Distance $t_{\min}$)
    $\mathbf{w} \leftarrow \mathbf{u} \times \mathbf{v}$
    $\det(\mathbf{A}) \leftarrow -\mathbf{r}_{\text{DIR}} \cdot \mathbf{w}$
    if $\det(\mathbf{A}) \neq 0$ then                ▷ Backface culling skipped for simplicity
        $\mathbf{o}_{\text{FR}} \leftarrow \mathbf{o}_{\text{RAY}} - \mathbf{o}_{\text{FAC}}$
        $d_A \leftarrow 1/\det(\mathbf{A})$
        $u \leftarrow ((\mathbf{v} \times -\mathbf{r}_{\text{DIR}}) \cdot \mathbf{o}_{\text{FR}}) \cdot d_A$
        $v \leftarrow ((-\mathbf{r}_{\text{DIR}} \times \mathbf{u}) \cdot \mathbf{o}_{\text{FR}}) \cdot d_A$
        $t \leftarrow ((\mathbf{u} \times \mathbf{v}) \cdot \mathbf{o}_{\text{FR}}) \cdot d_A$
        if $u \in [0,1] \wedge v \in [0,1] \wedge t > 0$ then
            if PointInPolygon(u,v) then
                HandleTransparentHit
                if $t < t_{\min}$? then
                    $t_{\min} \leftarrow t$
                    return true
                end if
            end if
        end if
    end if
    return false
end function
```

Whenever both conditions are met, we can keep track of a crossing. First, the ray in v-direction related to point $\mathbf{p}$ can only cross an edge when it is between the u-values of $\mathbf{p}_1$ and $\mathbf{p}_2$:

$$(u < u_{p_1}) \neq (u < u_{p_2})\,. \tag{2.68}$$

This condition ensures that p is strictly between the u-values of $\mathbf{p}_1$ and $\mathbf{p}_2$. Further, to find out whether $\mathbf{p}$ is above an edge or not, we calculate the edge's slope $m_E = (v_{p_2} - v_{p_1})/(u_{p_2} - u_{p_1})$ and compare the v-intercept for the linear equations for $\mathbf{p}$ and p_1 with the same slope m_E:

$$\text{POINT\_ABOVE} = \begin{cases} 1 & \text{if } m_E \cdot u_p - v_p < m_E \cdot u_{p_1} - v_{p_1}\,, \\ -1 & \text{else}\,. \end{cases} \tag{2.69}$$

The sum of these crossing values gives the winding number. When it is zero, a point lies outside the polygon, otherwise it lies inside. However, to account complex polygons e.g. polygons containing holes, the interpretation has to be slightly different. Here, only odd winding numbers account for points lying inside.

An alternative can be found with the Crossing number inclusion algorithm described by W. Randolph Franklin (1994). The reliability and numerical stability of many widespread point-in-polygon algorithms have been analysed by Schirra (2008). There exist a few edge cases that are not properly handled and could introduce numerical problems or ambiguities. This includes the proper classification of a hit on an edge, which are handled gracefully by some algorithms by balancing the distinction strictly e.g. labelling hits at the `left-bottom` to one and hits to the `right-top` parts to another facet. We found, that for Molflow these scenarios are rare enough to be negligible, hence we prioritise the performance-optimised algorithm described above.

2.3.6 Line-box intersection

Instead of using expensive polygon intersection tests in all cases, intersection tests can be simplified by testing for an intersection between a ray and a bounding box. We deploy a simple yet efficient line-box intersection algorithm called the SLABS METHOD, introduced by Kay and Kajiya (1986).

Given the ray described in vector form by $\mathbf{I} = \mathbf{o}_{ray} + \mathbf{r}_{\mathbf{dir}}t$ as in (2.50), we are looking for an intersection with an Axis Aligned Bounding Box (AABB), which can be defined by its minimal and maximal points $\mathbf{a}$ and $\mathbf{b}$ respectively. By looking at an AABB as a set of slabs, two lines parallel to the coordinate axis for every dimension, we can simplify this problem. With $\mathbf{I} = \mathbf{a}$ and $\mathbf{I} = \mathbf{b}$ we get the vector equations

$$\mathbf{o} + \mathbf{t}_A \cdot \mathbf{r} = \mathbf{a}\,, \tag{2.70}$$

$$\mathbf{o} + \mathbf{t}_B \cdot \mathbf{r} = \mathbf{b}\,, \tag{2.71}$$

where $\mathbf{t}$ represents intersection distances as a vector. Opposed to (2.50), here, we are interested in intersection distances for each dimension (x, y, z). We simplify the ray origin $\mathbf{o}_{ray}$ and the direction $\mathbf{r}_{dir}$ to $\mathbf{o}$ and $\mathbf{r}$, respectively. Next, we can find the intersection distance for each dimension by solving for $\mathbf{t}$ in each coordinate:

$$\begin{aligned} t_{Ax} &= \frac{a_x - o_x}{r_x}, \quad t_{Bx} = \frac{b_x - o_x}{r_x}, \\ t_{Ay} &= \frac{a_y - o_y}{r_y}, \quad t_{By} = \frac{b_y - o_y}{r_y}, \\ t_{Az} &= \frac{a_z - o_z}{r_z}, \quad t_{Bz} = \frac{b_z - o_z}{r_z}, \end{aligned} \tag{2.72}$$

where t_{Ax}, t_{Ay}, t_{Az} are the intersection distances with the minimal points $\mathbf{a}$ and t_{Bx}, t_{By}, t_{Bz} are the intersection distances with the maximal points $\mathbf{b}$ in each dimension x, y, z, respectively. As noted previously, the calculation of (2.72) can be optimized by precomputing the inverse ray direction. In the case that any of the ray direction components $r_{x,y,z}$ is zero, the inverse can be assumed to be 0 to handle the special case.[8]

Then, to determine whether the ray intersects the AABB, you check for overlap in the intervals $I_i = [t_{Ai}, t_{Bi}]$ in each dimension $i = \{x, y, z\}$. We require the greater value corresponds to the intersection with the minimum plane A and the smaller value to the intersection with the maximum plane B, as follows:

$$t_{min,i} = \begin{cases} t_{A,i} & \text{if } t_{A,i} > t_{B,i}\,, \\ t_{B,i} & \text{else}\,, \end{cases} \tag{2.73}$$

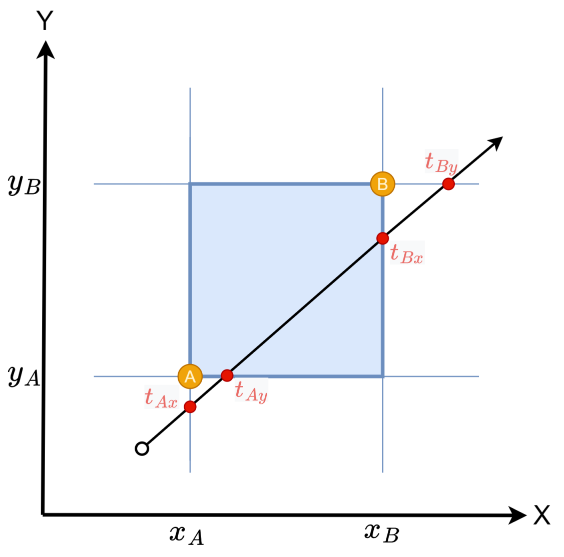
$$t_{max,i} = \begin{cases} t_{A,i} & \text{if } t_{A,i} < t_{B,i}\,, \\ t_{B,i} & \text{else}\,. \end{cases} \tag{2.74}$$

These become our $t_{min,i}$ and $t_{max,i}$. Then we check for all three dimensions to find the largest t_{near} value and the smallest t_{far} value. Doing so sequentially, we first compute $t_{min,x}$ and $t_{max,x}$. Then we compute $t_{near,xy}$ and $t_{far,xy}$ only concerning components from the x-dimension and y-dimension:

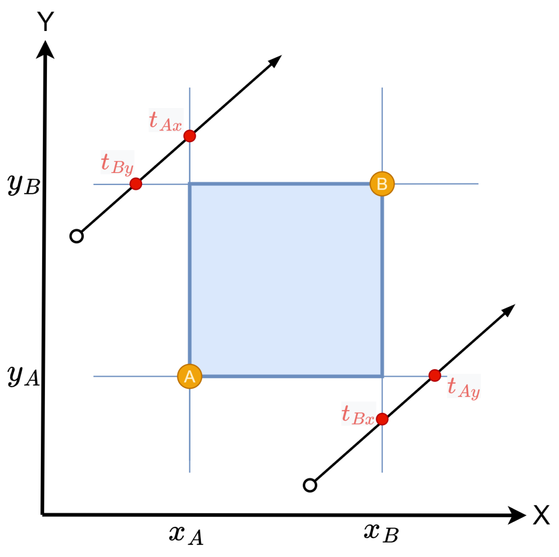
$$t_{near,xy} = \begin{cases} t_{min,x} & \text{if } t_{min,x} > t_{min,y}\,, \\ t_{min,y} & \text{else}\,, \end{cases} \tag{2.75}$$

$$t_{far,xy} = \begin{cases} t_{max,x} & \text{if } t_{max,x} < t_{max,y}\,, \\ t_{max,y} & \text{else}\,. \end{cases} \tag{2.76}$$

[8]Ideally the compiler can handle the division by 0 scenario gracefully.



(a) w/ intersection



(b) w/o intersection

Figure 2.11: Ray-box intersections with different near and far values for t_A and t_B. Min coordinate is labeled as A. Max coordinate is labelled as B. In 2.11a the ray intersects the box. In 2.11b no intersection occurs.

And lastly we compute:

$$t_{near} = \begin{cases} t_{near,xy} & \text{if } t_{near,xy} > t_{min,z}\,, \\ t_{min,z} & \text{else}\,, \end{cases} \tag{2.77}$$

$$t_{far} = \begin{cases} t_{far,xy} & \text{if } t_{far,xy} < t_{max,z}\,, \\ t_{max,z} & \text{else}\,. \end{cases} \tag{2.78}$$

There is an overlap – and therefore an intersection with the xy-plane of the box – if $t_{far,xy} \geq t_{near,xy}$, else the ray is not passing through the plane. Lastly, if $t_{far} \geq t_{near}$, the ray intersects the bounding box. Further, we require $t_{far} > 0$ (and inherently $t_{far,xy} > 0$), to ignore facets at the back of the ray with respect to its origin.

In figure 2.11a we can see a ray hitting the plane, because $t_{far,xy} > t_{near,xy}$ is true with $t_{far,xy} = t_{B,x}$ and $t_{near,xy} = t_{B,x}$. For figure 2.11b there is no hit, where the values are $t_{far,xy} = t_{B,y}$, $t_{near,xy} = t_{A,x}$ for the one case and $t_{far,xy} = t_{B,x}$, $t_{near,xy} = t_{A,y}$ for the other.

Ray-box intersections are fundamental to accelerate ray tracing routines. AABBs are used as building blocks for bounding volume hierarchies, which are elaborated in great detail in chapter 2.4.1.

2.3.7 Acceleration Techniques

Ray tracing describes a line-object intersection problem, where a naive algorithm would check for intersections with every single object to find the one with which it will intersect first. Most applications that deploy ray tracing algorithms work with geometries containing thousands to millions of polygons. Ray-tracing-based Monte Carlo simulations such as those conducted with Molflow, run enormous amounts of intersection tests to achieve accurate estimations. Hence, it is of great interest to optimise the algorithm with application specific techniques.

There are three main strategies to accelerate ray tracing algorithms, as outlined in Arvo and Kirk (1989): faster intersections, ray reductions, and ray generalizations. The most common approach is to speed up intersections. This is achieved by using acceleration data structures that group geometric primitives, allowing for more efficient initial intersection tests. The concept of *ray reduction* does not align well with Monte Carlo algorithms. Reducing the number of Monte Carlo events typically leads to less accurate or overly smoothed results. Techniques like sampling of results or terminating rays early can distort the physical accuracy of simulations, such as those in Molflow. These methods might provide smoother preliminary results but do not accurately reflect the underlying physics. Finally, generalizing or accumulating rays by replacing them with different shapes, such as beams or cones, is not effective for Molflow. This is because in Molflow, particles follow completely random trajectories, as described by equation (2.11).

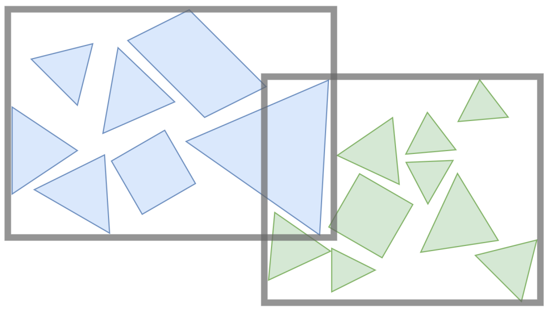
Most of the state-of-the-art techniques are in particular made for ray tracing as a rendering technique in graphical environments. The ray tracing problem in Molflow has different properties, which have to be taken into account to find and develop a suitable approach to accelerate the algorithm. For example, simulations in Molflow compute results for a *static geometry* for a *long period*, in contrast to typical rendering scenarios.

2.4 Acceleration Data Structures

This chapter aims to elaborate Acceleration Data Structure (ADS), according to the ideas elaborated by Pharr, Jakob, and Humphreys (2017). Good ray tracers deploy an ADS to reduce the computational complexity logarithmically, where expensive intersection tests against 3d primitives are reduced as much as possible.

A naive approach for ray tracing would consist of testing all polygons to find out the nearest intersection with the ray, which is obviously not feasible in most use cases for ray tracers like computer graphics or Monte Carlo simulations. Hence, ADS are maybe the most important technique for a good ray tracing engine as they can also be considered as the frame for utilising further acceleration techniques.

With a particular focus on graphical problems most ray tracing APIs (see NVIDIA OptiX[9] and Intel Embree[10]) are using so called Bounding Volume Hierarchies (BVH) as the default acceleration data structure. BVHs belong to the class of object partitioning structures. Meister et al. (2021) attribute this to a "predictable memory footprint", a "robust and efficient query" and a "scalable construction". A different type of ADS are KD-trees, which belong to the class of spatial subdivision structures. Unlike BVHs, KD-trees will subdivide the geometry not into individual objects, but it will subdivide the space, which makes it possible for individual objects to reside in multiple leaves. KD-trees have the advantage that they are usually said to be advantageous for static geometries where longer pre-processing times can be tolerated. Figure 2.12 shows how a scene containing geometric primitives is divided by a BVH and a KD-tree. We show how the same primitive scene is divided by a BVH (figure 2.12a) and by a KD-tree (figure 2.12b). For the BVH, here, the sub nodes created by the splits create spatial overlaps between the nodes. The KD-tree split leads to both nodes containing the same primitive.


(a) BVH

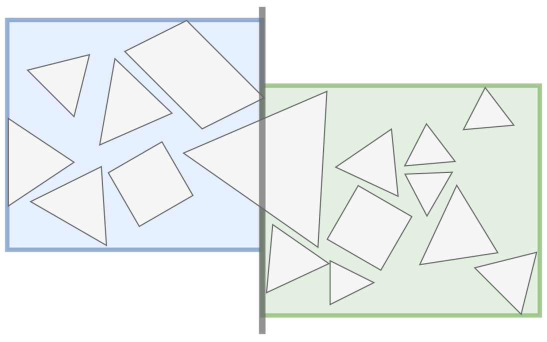

(b) KD-tree

Figure 2.12: Sketch of different acceleration data structures for the same scene of primitives. Left, the scene is represented by a BVH. One volume contains the blue primitives, the other contains the green primitives. The bounding volumes for the sub nodes overlap. Right, the scene is represented by a KD-tree. The spatial split divides the scene in two partitions: blue and green. It leads to a single primitive spanning both partitions.

[9]Accessed on 05/12/2023: `https://developer.nvidia.com/rtx/ray-tracing/optix`
[10]Accessed on 05/12/2023: `https://www.embree.org`

2.4.1 Boundary Volume Hierarchy

The concept of a bounding volume hierarchy is as follows. Given a set of geometric primitives, we construct a hierarchical tree structure. The leaves of this tree store references to individual primitives, and the internal nodes contain pointers to their child nodes. Each node in the tree is associated with a bounding volume that encompasses all of its child elements. Bounding volumes eliminate the need for individual ray-primitive intersection tests at the internal node level, replaced by a single more simplified intersection routine that depends on the type of bounding volume. At leaf level, a final intersection test with a bounding volume has to pass, before all primitives are tested for individually. There are different types of bounding volumes, where axis-aligned bounding boxes (AABBs) are most commonly used. AABBs are cuboids, whose axes are parallel to those of the global coordinate system. This is opposed to oriented bounding boxes (see Shirley, Wald, and Marrs, 2021), whose edges axes do not have to be axis-parallel. Further techniques exist, such as bounding spheres or axis-aligned bounding tetrahedra or octahedra[11 ,12]. Their effectiveness is largely dependent on the utilised geometry and has not been confirmed for arbitrary geometries yet. Hence, for further investigation we focus only on AABBs. Extensive research into BVH techniques has been conducted and summarized by Meister et al. (2021) in their survey.

Efficient ray-box intersection

When using AABBs as bounding volumes for the inner nodes of a BVH, we can use the so called `slabs algorithm` (Kay and Kajiya, 1986) for a ray-box intersection test, which has been deployed for Molflow's ray tracer. A visualisation of the algorithm is sketched in figure 2.13. The complete description of the algorithm is given in chapter 2.3.6. Majercik et al. (2018) give a modified version – tuned for vector instructions – of the algorithm as sketched in listing 2.1. The algorithm works by first determining the intersection distance t between the ray and the lower (min) and the upper bounds (max) of the AABB on each of the 3 axes. With the corresponding t values, we can define a set of cases where a ray does not intersect with a given bounding box. These cases are sketched in figure 2.11.

BVH in Molflow 2.7

Historically, Molflow deployed a BVH with AABBs for performant ray tracing-based simulations. Most importantly, a complete revamp of the memory layout of the data structure and the splitting criterion achieved major performance improvements,

[11]Accessed on 20/12/2022: `https://medium.com/@bromanz/axis-aligned-bounding-tetrahedra-and-octahedra-for-ray-tracing-bounding-volume-hierarchies-683751d84bca`

[12]Accessed on 20/12/2022: `https://github.com/bryanmcnett/aabo`

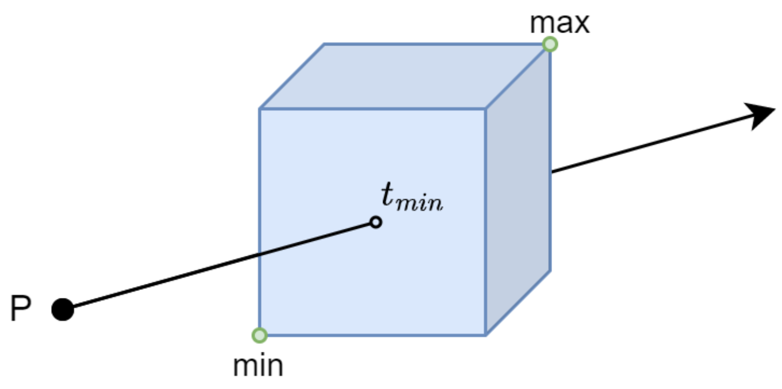


Figure 2.13: Visualisation of the ray-box intersection algorithm. t_{min} denotes the minimal intersection distance for the box with the minimum and maximum bounds min and max respectively.

```
bool slabs(vec3 p0, vec3 p1, vec3 rayOrigin, vec3
   invRaydir) {

        vec3 t0 = (p0 - rayOrigin) * invRaydir;
        vec3 t1 = (p1 - rayOrigin) * invRaydir;
        vec3 tmin = min(t0,t1), tmax = max(t0,t1);

        return max_component(tmin) <= min_component(
           tmax);
}
```

Listing 2.1: Efficient slab test algorithm given by Majercik et al. (2018). To neglect conditional statements, the algorithm makes use of possible vector instructions.

which will be discussed in more detail in chapter 7.3.2. At node level primitives are contained in minimal AABBs. Following a top-down approach, AABBs are split depending on a specified splitting criterion (see 7.2), where a parent node contains all its descendants and is essentially a union of their AABBs. Each inner node of such a BVH has M children, where usually binary trees are deployed ($M = 2$). The leaf nodes contain up to P primitives. Primitives are usually tested first by attempting a simplified intersection test against their individual bounding box and only then against the strict primitive description (e.g. polygon or triangle). With this, the number of intersection checks can be minimised for N facets from $O(N)$ to $O(\log_M N)$ in practical cases, due to the early rejection of subsets[13].

At the beginning of this study, for Molflow the optimised BVH implementation allowed for a minimum of 8 facets per box (`MINBB`$= 8$) and a maximal tree depth of `MAXDEPTH`$= 50$, which proved to be a good choice using a simple tree splitting approach that has been deployed in Molflow 2.7. For the construction of a BVH, a simple top-down approach has been used using median-based splitting. It starts by computing a single Bounding Box (BB) containing all facets of a structure, which represents the starting node at root level. That node is then cut by the following criterion:

1. Calculate the center coordinates (3d-coordinates) of the Bounding Box.
2. Calculate difference between the number of elements in one or the other side of the center, for each axis (x, y, z):
   a) Split a box in the center of each axis.
   b) A facet is in one side if the corresponding coordinate of its centroid is lower than the center of the box, and on the other side if it is greater.
3. Make a cut at the center of that axis, where the difference is the lowest (striving for an equal distribution).

Primitives are then distributed in two new nodes `rightNode` and `leftNode`, depending on whether the center lies on one side or the other of the cut-axis. The newly created nodes are then further divided recursively by the same scheme until either:

- a maximal tree depth `MAXDEPTH` has been reached.
- the amount of children in a single node after a cut would be lower than `MINBB`.

As part of this thesis, we implemented state-of-the-art techniques and developed new methods that are more efficient for partitioning BVHs. These algorithms are elaborated in great detail in chapter 7.2.

[13] Accessed on 20/12/2022: `https://www.scratchapixel.com/lessons/3d-basic-rendering/minimal-ray-tracer-rendering-simple-shapes/ray-box-intersection`

2.4.2 KD-tree

In contrary to BVHs that use object partitioning to divide nodes, the KD-tree creates partitions by dividing the space using split planes and storing these planes in the inner nodes. Unlike the general binary space partitioning tree (BSP) that allows split planes that are arbitrarily oriented, the KD-tree only allows splits parallel to the three axes. Figure 2.14 visualises the splits of a KD-tree for an exemplary geometry in Molflow. Opposed to BVHs, primitives can belong to both partitions,

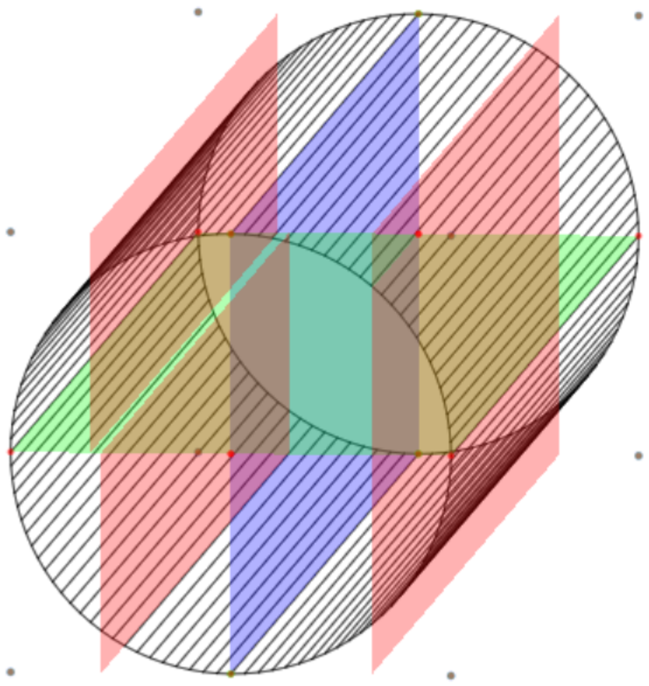


Figure 2.14: Visualisation of a KD-tree for an example Molflow geometry. Spatial splits are coloured hierarchically (blue, green, red). First, the geometry is split by the blue plane in two nodes. Next, these nodes are further split by the corresponding green splitting planes. These nodes are further divided by the splitting planes colored in red.

when they straddle the split plane. This gives a bigger memory overhead, even when the primitives are just stored giving a reference. KD-trees have the interesting property that the ray-tracing algorithm can be terminated early, when the first hit location has been identified[14]. This is because intersecting a KD-tree can be done

[14]This is not entirely true for Molflow, when transparent facets are used.

in front-to-back order, so initially we start with the potentially closest hit location. Since a KD-tree is thus *spatially aware*, this property can further be used e.g. to quickly navigate between neighbouring sub-volumes and to restart the search from a given node that contains the starting point.

Traversal techniques

There are many possible traversal techniques for a KD-tree used for ray-tracing applications. They mainly differ in their approach to restart when an intersection test in a lower level node leads to no intersection. Lira dos Santos, Teichrieb, and Lindoso (2014) describes different methods and discusses their advantages. Here, we focus on the following techniques, both sketched in figure 2.15.

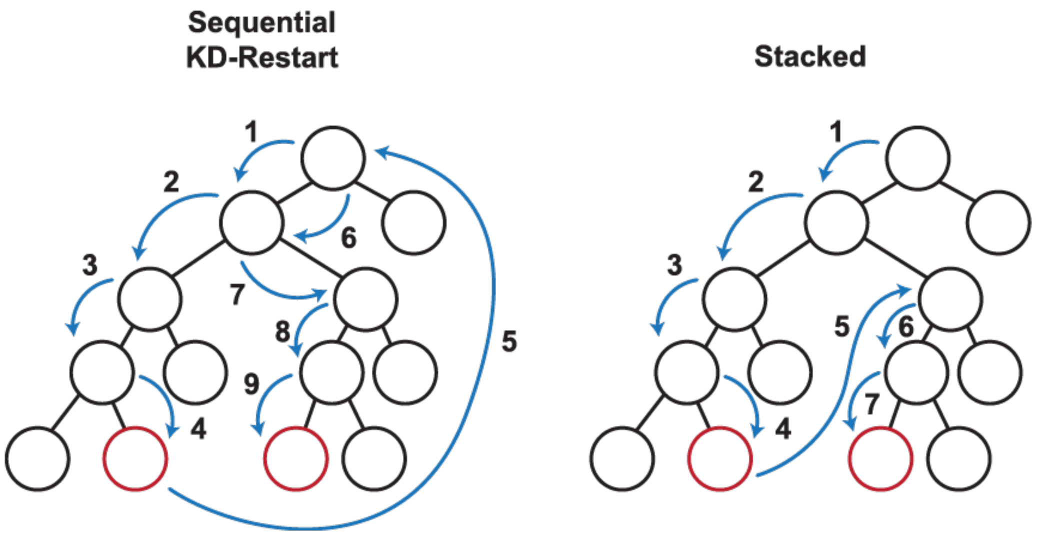


Figure 2.15: Naive and stacked KD-tree traversal (Lira dos Santos, Teichrieb, and Lindoso, 2014).

Sequential KD-Restart A naive implementation for a traversal algorithm for KD trees is the kd-restart traversal. In this method, at each node visited during traversal, the algorithm performs intersection tests to determine whether the current node's bounding volume intersects with the ray or query being processed. When no intersection is detected for an inner or leaf node during traversal, the traversal algorithm always resets and restarts from the root node. This iterative process continues until a potential hit location is identified, often resulting in repeated traversal of previously visited nodes. A downside of this technique is encountered for KD trees that are deep or contain many empty regions, which may lead to repetitive traversal of the same branches.

Stacked The de facto standard traversal algorithm for navigating KD trees efficiently. It involves maintaining a stack to track nodes encountered during traversal. This allows the algorithm to backtrack effectively, especially when two inner child nodes are intersected. The stacked nodes are revisited once all child nodes have been processed. The stacked traversal method follows a front-to-back approach, prioritizing nodes closer to the ray origin for processing. It can be implemented using either recursive or iterative techniques. The stacked approach enhances performance by minimizing redundant traversal and efficiently managing node exploration.

Another technique that uses links between bottom-level nodes or quick traversal is presented in section 7. This algorithm has an interesting property for Molflow's use case making use of static geometries and spatially close reflection points.

2.4.3 Grid-based partitioning

Uniform grids as an acceleration data structure for ray-tracing introduce some interesting properties. The space is subdivided in equidistant intervals for each dimension, resulting in equally shaped boxes or so called voxels (volumetric elements). Primitives are then distributed to belong in one or more voxels, in case of an overlap. To find intersections with a primitive, following the ray origin the grid can be traversed in front-to-back order. For each voxel, intersection tests with the individual primitives are carried out until an intersection can be determined.

The performance of ray-tracing with a uniform grid obviously depends on the chosen resolution, where the optimal amount of voxels is roughly proportional to the amount of primitives n. This can be achieved in most cases by using a uniform grid resolution in relation to the amount of primitives $r \in \mathbb{N}$ with $r \approx c\sqrt[3]{n}$ (Ize, Shirley, and S. Parker, 2007). Best case, an intersection can be found in a few steps giving a traversal complexity of $O(1)$. If no intersection can be found at all, Ize, Shirley, and S. Parker (2007) give a worst case complexity of $O(r = \sqrt[3]{n})$ with the proposed optimal resolution, when a ray is traced parallel to one of the coordinate axes. The efficiency for a uniform grid can mainly be scaled with the resolution. A high resolution also results in an increased memory consumption. For Molflow, uniform grids have the interesting property, that they could potentially be used to introduce the feature to simulate intermolecular collisions in the viscous regime (Kn < 1). As other data structures promise better performance, grid-partitioning has not been investigated further in this work.

2.5 State of the art - Ray tracing

There exist various ray tracing libraries that encompass high-performance kernels for both CPUs and GPUs. Embree from the Intel Corporation (Wald, Woop, et al., 2014) might be the most established choice for CPU-based simulations in particular for rendering applications, such as V-Ray, Autodesk RapidRT and blender. The open-source framework delivers "hand optimised, low-level kernels" – in particular via directly implemented vectorisation – and is suitable for a variety of different workloads. Shriwise (2018) utilised Embree for the Direct Accelerated Geometry Monte Carlo (DAGMC) toolkit, a simulation software for Monte Carlo Radiation Transport. Hence, DAGMC shares similar constraints for a ray tracing engine with Molflow. A downside of Embree for physical simulations is the inherent use of single precision ray tracing calculations[15], which the author of the paper has targeted to fix with his double precision interface double-down[16]. The inherent idea was to create a mixed precision BVH, where the original single precision calculations are used for intersection tests with bounding boxes and double precision is used for potential intersections with primitives.

Another ray tracing library targeting both CPU and GPU[17] is the code pbrt (Pharr, Jakob, and Humphreys, 2017). pbrt is a ray tracing rendering code with implementations of BVHs and KD-trees keeping state-of-the-art techniques in mind, also on the cache-optimisation level. It is straightforward to modify and extend the pbrt implementation with custom routines. This is why we decided to use pbrt's ADS implementation as the foundation for Molflow's revamped ray-tracing engine compared to other ray tracing libraries. The open source code, using a BSD-2 license, is ideal to use for extensive research for Molflow's ray tracing backend. The code enables us to focus on implementing new techniques, while serving as a great foundation with performant implementations for standard rendering techniques.

The study by Roberto Kersevan and Pons (2009) found that the use of graphic computing units (GPUs) to improve Molflow's Monte Carlo algorithm is constrained, primarily due to limitations in the foundational ray-tracing algorithm. With the advance in technology, namely the increase in performance of GPUs in general and especially the introduction of NVIDIA's RTX graphic cards, one might expect that porting the Molflow algorithm to run on GPUs will allow for the necessary performance of more demanding and long term simulations need. For that matter, I reevaluate the GPU acceleration study conducted for Molflow (Roberto Kersevan and Pons, 2009), with today's hardware and software solutions. Furthermore, I study the possibility of making use of the RT cores from NVIDIA's Turing GPUs. They are said, as stated by NVIDIA (2018, p.30), to accelerate "Bounding Volume

[15]detailed explanations in chapter 8
[16]Accessed on 04/06/2022: `https://github.com/pshriwise/double-down`
[17]GPU support officially part of the source code in version 4.

Hierarchy (BVH) traversal and ray/triangle intersection testing (ray casting) functions", which is not how the ray-tracing algorithm in Molflow works right now. We have to see if the AABB tree with the Line-plane intersections will benefit at all from NVIDIA's new raytracing pipeline.

We conduct an initial study to investigate first, if the major improvements in general performance from common consumer-class GPUs are enough to justify porting the Molflow algorithm to GPUs. Afterwards, we take a proper look at the current and future generation of NVIDIA's RT core GPUs and see if using consumer-class Turing graphic cards allow for a proper boost in performance and if the usage or investment in dedicated HPC hardware, in specific Tesla T4 or newer, would be worth it.

2.6 Contributions

As part of the thesis, the following scientific contributions have been made. In the context of ray tracing simulations for physical models, with Molflow as the leading example, an in-depth analysis of state-of-the-art acceleration data structures and corresponding splitting techniques has been carried out. This led to the development of a highly specialised ray tracing engine based on KD-trees that is constructed with a new splitting heuristic called Hit Rate Heuristic (HRH), which is introduced in section 7.2.4.

The suitability of a GPU kernel that is fully leveraging the potential of hardware-accelerated ray tracing units, exemplary on NVIDIA RTX GPUs, has been analysed. We developed and demonstrate a technique called Neighbour Aware Offset (NAO), that counters the negative effects of the single-precision calculations, while using the full RT pipeline of the before-mentioned hardware.

We investigated several stopping criteria for Monte Carlo simulations and evaluated the usage of various parameters most suitable to determine the general convergence of a solution.

A rework of the data structures and algorithms supporting the time-dependent simulations has been conducted, leading to large speedups with a minimal addition to the constraints.

Further, with a focus on the User Experience (UX), several algorithms that are used during the pre-processing steps or the design phase have been reevaluated and improved based on the specific data structures that are employed in Molflow and Synrad.

3 Software Development Process

Up to the point of starting this PhD thesis, the main Molflow development process has been performed by only a single person, where the primary focus is on functionality and stability. Therefore, the aspects of software engineering need to be reviewed so that vital components are in focus and can be easily changed without affecting too many parts of existing software. Initially, the codebase was shared with the Synrad project, prioritisation of Molflow's development, and a major overhaul of the underlying software architecture led to diverged code, and thus increased software development costs.

In the initial phase of this thesis, Molflow has been analysed with appropriate profiling tools to determine relevant components of the simulation engine that have a big computational impact and give the potential for an increase in overall performance (see chapter 3.6). Further, the necessity to bring Molflow and Synrad back to a converged codebase, as they share large parts of the code, is elaborated with a structured plan that led to an upgraded simulation kernel for both tools, further enabling the usage in HPC environments. The base version achieved with this thesis is Molflow 2.9.5 , which introduced a strong separation between the user interface and the simulation kernel. A dedicated CLI-based version MOLFLOWCLI has been released, that enables users to run Molflow simulations in headless environments, such as MPI clusters, or to utilise it for more advanced simulations within scripts. Due to some noticeable problems in the early releases, the incremental versions for Molflow 2.9+ have been flagged as BETA up to the submission of this thesis. This state has been kept mainly due to the lack in resources to validate and verify all components that have undergone larger rewrites. Further modifications have been made as part of an independent release[1], solely to drive and finalize some of the features or their corresponding tools for empirical evaluation. Most noteworthy is the availability of convergence-based stopping criteria – that have been implemented, but not been enabled in the official source code – and further adjustments to the developed GPU kernel.

[1]Accessed on 05/12/2023: `https://github.com/iBaer/molflow-phd-fork`

3.1 Code Refactoring

Molflow 2.7 , the previous baseline, included a major revamp building on top of Molflow 2.6 , moving the simulation engine from a master-worker-based approach using subprocesses to a thread-based approach. While this helped in reducing the memory requirements due to shared resources among all simulation threads, it was observed that the newly developed data structures were deeply entangled with the graphical user interface. Furthermore, due to the complex nature of these changes and limited human resources, they were not implemented in Synrad, resulting in diverging codebases. After considering the clearly defined goals of this thesis, in particular, the development of a CLI application for HPC environments and a GPU-based simulation kernel, a rollback to Molflow 2.6 has been favoured and conducted in the initial phase of this work. This process included a merge of all changes excluding the architectural revamp of the simulation engine, leading to a new base version on which the work on this thesis builds upon, Molflow 2.8 . Refactoring is an important step for the Molflow code base. In order to achieve code unification for Molflow and Synrad, a CLI-based application and a GPU kernel, the separability and independence of the user interface and the simulation code are a necessity. This led to a general separation of the simulation modules, which have been sketched in figure 3.1. The `SimulationManager` creates a common interface for operating the simulation kernel. This enables both the GUI and CLI applications to initialize data structures for the input geometry, and to launch and monitor simulations. `SimulationCore` is a concept that easily integrates different simulation kernels, where the GUI and CLI only need minimal information about the actual simulation, which could use the CPU or GPU kernel.

Initially, to ensure that as part of the refactoring process no functionality-breaking changes are made, a proper test suite had to be set up. Due to the strong coupling to the GUI and low cohesion of the available classes writing manageable unit tests was not possible. As a workaround, the following approach has been used to automatically validate the correctness of the simulation kernel. To achieve this, the general idea was to first create a gold standard for the results from a set of input geometries. Simulation results from newer development versions of Molflow could then be generated and validated against this gold standard. An automatic test suite was set up in Python with the GUI automation library PyAutoGUI[2] to automatically run new simulations and save the results. The first versions of the script run simulations with a fixed amount of particles, where the simulation kernel worked only with one thread and with a fixed random seed to enable reproducibility. This was later changed to an approach to validate simulations only against a set error threshold. A full description of the algorithmic changes and adapted data structures

[2]Accessed on 02/12/2022: `https://pyautogui.readthedocs.io/en/latest/index.html`

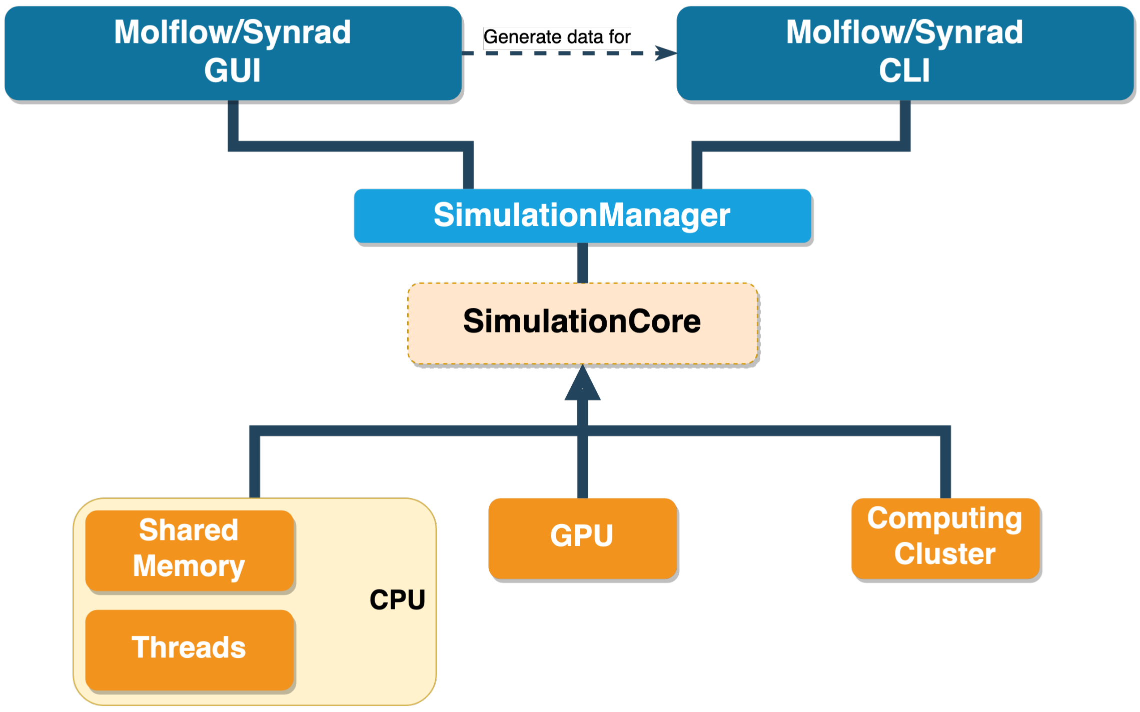


Figure 3.1: Sketch of the proposed design of the dependencies for Molflow's distinguished components, where the so-called `SimulationManager` is the common interface for both GUI and CLI applications to launch simulations, based on a pre-specified simulation kernel implemented via the `SimulationCore`.

for the code base of Molflow and Synrad is described in chapter 5. The revamped software architecture was a cornerstone for the study of the thesis, starting with the implementation of the CLI.

3.2 Molflow CLI

A critical component for the development process was the design of a Command Line Interface (CLI) version of Molflow, which offers new possibilities in efficiently running and testing simulations. The CLI gives Molflow users the flexibility to run simulations without the graphical user interface. With the CLI, simulations can be run even on dedicated compute resources, which are often headless servers, such as those in HPC environments. For us, this is a necessary condition to integrate MPI for parallel and distributed computations, which is further elaborated in chapter 3.4. A set of input arguments gives users the flexibility to launch simulations with varying parameters intuitively. It is possible to run simulations with various end conditions, such as a time limit or a desorption limit. Global simulation parameters or facet parameters can be changed regardless of the given input file. The possibility to couple a set of simulations gives users a powerful tool in rapidly analyzing multiple runs.

Following Molflow 2.7 , the development of the CLI demanded major changes to the software architecture as elaborated before. Simulation data structures were deeply connected to the graphical user interfaces. This created a lot of dependencies and resulted in a large overhead for simulations. By decoupling the simulation engine, as sketched in figure 3.1, and with the revamped design and the removal of dependencies, we were able to create an automatic testing infrastructure for Molflow. This infrastructure helped us quickly identify and fix bugs, ensuring a more reliable software.

Overall, the CLI has been an important addition to Molflow's tool set. It increases the audience to classical HPC environments and further leverages scalable compute resources. Simulations and following analysis has become more flexible and customizable for specific needs. Molflow's CLI has already been proven in scenarios such as the development of iterative simulation using a script approach conducted by Henriksen, M. Ady, and R. Kersevan (2023).

3.3 Automated testing and GitLab CI/CD

Molflow's source code environment has been using an internal GitLab server for version control. In the early development stages of this thesis, this environment has been extended by GitLab's CI/CD (Continuous integration and continuous delivery) functionality. CI/CD enables development teams to focus on deliverables, while leading to better software quality. The idea of CI is that code changes are saved more frequently to the version control repository and validated. Molflow's CD extends the environment with an automatic multi-OS build system to release new software versions. Officially, the following operating systems are directly supported and provided:

- Windows 10 ,
- macOS Monterey (x86_64) ,
- macOS Monterey (ARM) ,
- Ubuntu 20.04 LTS (Debian binary) ,
- CentOS 8 (Fedora binary) .

3.3.1 Test suite

With the release of Molflow's CLI controls, a test suite had been developed and integrated into the GitLab CI/CD pipeline, testing various features for functionality and simulations for integrity and performance. Function tests have been progressively added to the test suite for new or refactored functions. This guarantees functionality also against edge cases. Further, the application is validated by performing simulations on a set of more than 10 different test cases, that include most of the possible geometrical properties and simulation parameters. Initially, a gold standard has been generated with one of the earlier versions that has been verified manually. Simulations are carried out for all test cases for the new build for a fixed time. The results are then validated against the gold standard by applying a different threshold depending on the counter type (global, per facet, per texture, etc.). The test is run multiple times for each test case and has to return a positive result in multiple attempts in a row. We found that this validates the stochastic nature of the Monte Carlo simulation more accurately without returning potential false negative reports.

Another part of the test suite runs performance tests in order to observe potential performance degradations. Simulations are run for a longer period to account for potential outliers. Our implementation keeps track of the best results from 20 commits, used to manually identify significant changes related to a particular commit. For each build we run 5 simulations and check performance values (MC events per second) against the set of best results. As we are expecting a wider spread for performance results in the testing environment, we evaluate multiple quantities. For each build, we track the best performance υ_{max} and the median performance υ_{med} across all 5 runs. For the performance values of the best run in the tracked set $\upsilon_{max,best}$ and $\upsilon_{med,best}$, the performance test is passed if any of the following conditions is true:

$$\upsilon_{max} > 0.95 \cdot \upsilon_{max,best} \,, \tag{3.1}$$
$$\upsilon_{med} > 0.95 \cdot \upsilon_{med,best} \,, \tag{3.2}$$
$$\upsilon_{max} > \upsilon_{med,best} \,. \tag{3.3}$$

Throughout the development, we found that this approach was effective in identifying performance degradations.

3.3.2 Gitlab pipeline

Public releases passing all sufficient stages of the pipeline could then be published with the incremental in-source changelog. The pipeline of the Gitlab CI/CD is sketched in figure 3.2. The main code base – either Molflow or Synrad – together with the shared code base are used for the CI stages. First, some parameters are fetched and parsed such as the rolling changelog. Next, the code is built and on success tested in two phases: When functionality tests have passed, the code is benchmarked and checked against performance degradation compared to previous builds. On success and in case of a public release, the multi-OS builds will be packed with all necessary files, uploaded and published as part of a GitLab release entry with an automatic changelog.

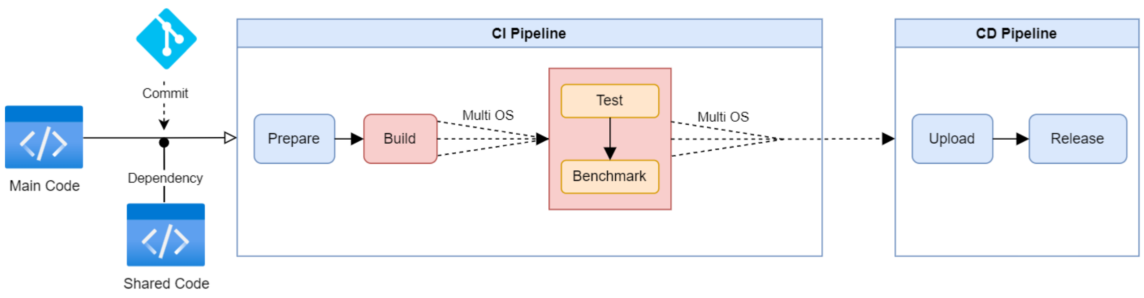


Figure 3.2: Sketch of the GitLab CI/CD pipeline. A commit to the main code triggers the CI pipeline, where a tagged release will further trigger the CD functionality for a successfully tested build.

3.4 Parallel and distributed computing

As Molflow's TPMC method is *embarrassingly parallel*, we expect the application to scale quasi-linearly using MPI. The new implementation, that was achieved as part of this thesis, uses an OpenMP/MPI hybrid approach for parallelisation. OpenMP is used for local parallelilization and MPI is used to parallelize on multi-node machines: e.g. computing clusters. OpenMP was deployed to replace the old simulation engine using *phtreads*. It is highly portable and is supported on all targeted operating systems for Molflow: Windows, Linux and macOS. Furthermore, ease of use and its scalability proved to be good arguments to adopt OpenMP. It was straightforward to make good use of shared memory utilization and thread synchronization. Unfortunately, due to the many architectural changes, that came along e.g. an updated ray tracing algorithm, we can not make a direct statement to any improvements that are solely related to changing from pthreads to OpenMP.

We validated our simple MPI implementation, that enables Molflow to run on large MPI clusters such as CERN's HPC infrastructure (see appendix A), with some simple experiments using the MPI pipeline which is roughly sketched in figure 3.3. Transfers happen only when the input from the master node is shared with the other nodes ($M \rightarrow S$) or when the results are collected back to the master node ($S \rightarrow M$). Sharing and gathering of the data is done with serialized data structures

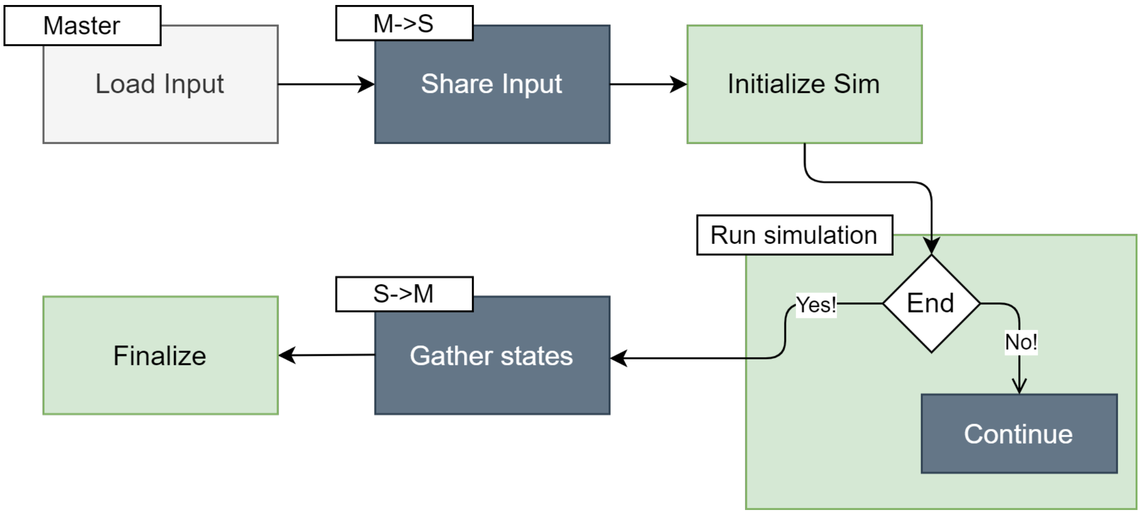


Figure 3.3: Pipeline showing the MPI transfers. After loading an initial input file, it is shared from the master node to the worker nodes. When the simulation on all worker nodes is done, the end results are gathered back on the master node.

via `MPI_Bcast` and `MPI_send/recv`, respectively.

For a 128 nodes system we were measuring a speed up of 123.5× for a simulation run of half an hour. This exemplifies that the speedup scales quasi-linearly as no communication is necessary during the actual simulation. Communication overhead is only added in the pre-processing and the post-processing steps. The capability to run Molflow on large computing clusters will allow users to run simulations with more complex geometries and simulation parameters.

3.5 OS x Compiler

Molflow's sources can be built officially on all major operating systems: Windows, Linux and MacOS. Windows and MacOS are mainly used in the community for their easy to deploy systems and are common among many CAD engineers and physicists at CERN. An initial study running the same simulation on the same hardware concluded that Windows with the MSVC compiler performed worse on average, than a Linux build with either a CLang-7 or GCC-8 build. Both were around 50-60% faster with compiler optimizations enabled, where the GCC build proved to be more stable in performance. The numbers are shown in figure 3.4. Hence, this was utilized as the main system for building new features such as the GPU accelerator and the MPI runtime.

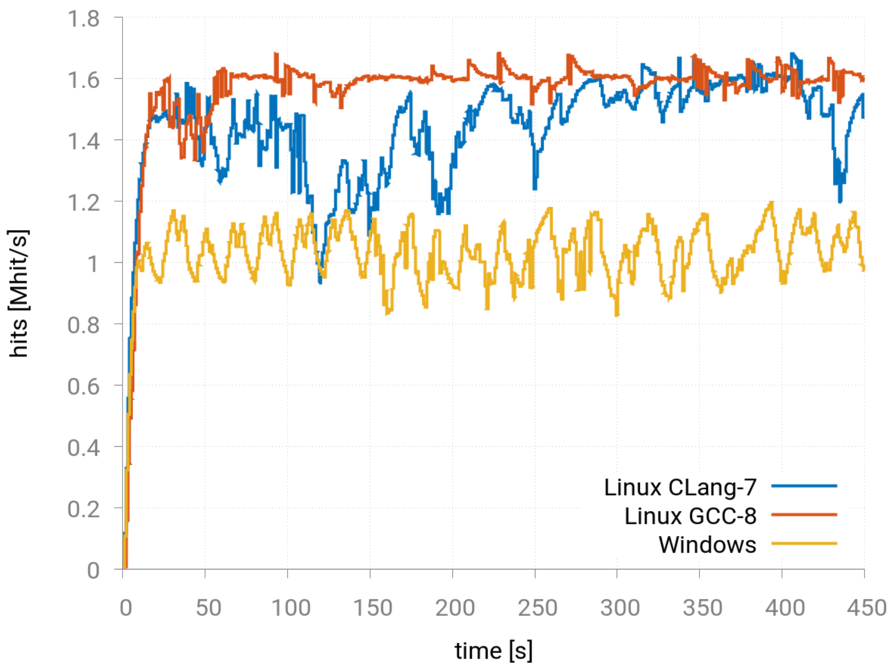


Figure 3.4: Performance is measured for a simulation with an arbitrary test case for 450 seconds. The benchmark is conducted for Windows, and Linux with GCC and CLang as compiler on the same machine configuration. The simulation on Windows is the slowest on average. The CLang build performs well, but has some sudden performance drops. Best average performance was achieved using the GCC build on Linux.

MacOS performance was excluded for this benchmark, as a direct hardware comparison was not feasible. Though, good performance with the newer architectures was found without any major adjustments[3].

3.6 Profiling

Profiling is essential in assessing the performance of Molflow's algorithms across various test environments, which helps with the identification and optimization of any potential inefficiencies within the software. Based on the main release, before the start of this thesis, Molflow 2.7.7 , the simulation kernel has been profiled with Intel Vtune Profiler[4]. The profiling includes all steps starting from pre-processing, that is the construction of the corresponding data structures, to the ray tracing algorithm and the following post-processing procedures including the accumulation of statistics and the initialization of new particle states.

In table 3.1 the profiling analysis for two geometries, which are further elaborated in chapter 4, is presented. Each simulation has been run for 6 minutes. Without getting into too many details about the test geometries and geometry properties for Molflow simulations, we give a brief overview as follows. The first test case (see figure 4.1) is a typical geometry for Molflow. It consists of just a moderate amount of facets with desorption or sticking properties. It mainly consists of reflective facets. On the other hand, there is the second test case (see figure 4.3), which consists mostly of absorbing, desorbing or teleport facets. For the first geometry, the ray tracing routines `Ray-BB-Intersection`, `Ray-Plane-Intersection` and `Point-in-Polygon` (elaborated in chapter 2.3.1) play a dominant role, accumulating to $> 75\%$ of the total runtime. While these routines still have a major impact on the overall performance for the second case ($> 40\%$), it mainly has to deal with the creation of new particles and thus with the selection of a source facet and location, the particles' direction, and the velocity. The main contribution for the overhead comes likely from the use of an outgassing map. The dot product plays a dominant role for a standard linear algebra routine, which mainly results from its frequent use in various functions. Not all relate to the ray tracing algorithm itself but the post-processing of its result. According to the profiling data about 78% of calls of the dot product from the first test case and about 82% from the second test case resulted from the `Ray-Plane-Intersection` routine. Thus, trying to enhance the performance of these parts of the simulation kernel, e.g. by processing the data via

[3]Fetched 28/05/2023:
`https://molflow.web.cern.ch/node/378`

[4]Fetched 29/11/2023:
`https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html`

BLAS (Basic Linear Algebra Subprograms) or in a SIMD (Single Instruction Multiple Data) pipeline, in general, is certainly of interest and could lead to significant speed-ups. The preprocessing step to create the Acceleration Data Structure did not play a dominant role at all, especially when considering the one-time construction for long duration simulations.

Function	**Time #1**	**Calls #1**	**Time #2**	**Calls #2**
Ray Tracing				
Ray-BB-Intersection	33.68%	7,079,123,496	18.95%	3,651,648,053
Ray-Plane-Intersection	28.13%	74,506,703	11.42%	31,659,495
Point-in-Polygon	14.04%	271,751,382	5.52%	131,471,346
Dot/Scalar product	4.34%	1,163,136,136	4.83%	713,553,272
Other MC steps				
Increase Counters	2.57%	192,672,446	4.92%	78,786,606
Time lookup	1.91%	74,509,070	0.78%	31,651,480
[Random] MT generate	1.34%	607,606,714	0.80%	587,957,246
[Random] MT double	0.44%	303,797,972	0.26%	293,914,726
Perform Bounce	1.10%	74,318,913	0.08%	6,478,337
Source selection	0.04%	189,603	38.98%	25,182,060
Binary search	No Outgassing Map		3.12%	25,182,754
Preprocessing				
Build AABB Tree	1.99%	3	0.52%	3
Construct BB	1.81%	3,791	0.65%	18,331

Table 3.1: Benchmark of multiple geometries for 6-minute simulations. Geometry 1: 1023 facets & Geometry 2: 4678 facets w/ teleport

Overall, the profiling gives plenty of insights for potential improvements, which largely steered the direction of this thesis. In particular, we put an emphasis on the enhancement of ray tracing routines in chapter 7 as well as the development of a GPU kernel for a different perspective in chapter 8. Complementary benchmarks further led to the investigation for improvements for the time dependent simulations in chapter 5.3.

4 Test cases

In this chapter we introduce the test cases that have been used for the benchmarks and validation tests. First, we describe some characteristics for polygon meshes that are used to provide context for the experiments. Further, some geometries are highlighted and described in more detail. Lastly, the computed characteristics for all tested cases used are given.

4.1 Geometry characteristics

In order to comprehensively describe the geometries utilized in our experiments, our aim is to represent the significant characteristics of each geometry. To achieve this, we draw from concepts introduced by Pharr, Jakob, and Humphreys (2017) and Akenine-Möller, Haines, and Hoffman (2008). We initially consider common characteristics associated with typical polygon properties. Additionally, we derive more intricate values to correspond with prevalent rendering or ray tracing properties. It is important to note that there is no single characteristic that can describe all aspects of a geometry and how it ultimately affects the ray-tracing performance in relation to different ADS and traversal techniques. In our analysis, we use the following key values:

- Number of polygons N_{fac}: Indicates the complexity of the geometry. Higher numbers may lead to lower hit rates as a result of more ray-primitive intersection tests. Lower numbers may lead to lower accuracy.
- Number of triangles N_{tri}: Lists the number of triangle ($n = 3$-polygons). Higher number, means less optimized for CPU.
- Number of rectangles N_{rect}: Lists the number of rectangles ($n = 4$-polygons). Our GPU ray tracer typically works with triangles, so all n-polygons with $N \geq 4$ need to be triangulated, which can increase the number of primitives in the scene.
- Number of $n > 4$-polygons $N_{n.poly}$: Lists the number of polygons ($n > 4$-polygons) that are neither triangles nor rectangles. Similar to N_{rect}, this metric highlights additional computational demand for properties that are otherwise native to $n = 4$-polygons but not $n > 4$-polygons, such as texture mapping.

- Number of vertices N_{vert}: Indicates the complexity of the geometry similar to the number of facets.
- Average polygon vertex count $\bar{N}_{vert}$: Higher values imply more complex polygons.
- Median polygon vertex count $N_{medvert}$: Indicates based on the median what is the average polygon type in the geometry.
- Median polygon area $\tilde{A}_{polygon}$: Provides a measure of the average polygon size. Smaller polygons may increase the chance of ray misses, requiring more traversal in acceleration structures like BVHs or KD-trees, particularly in larger scenes where the density of polygons might be higher.
- Bounding sphere radius r_{bound}: Gives an idea of the size of the geometry. Larger geometries may require more traversal steps in acceleration structures and may lead to longer rendering times. The radius is computed by

  $$r_{bound} = \max_i \left(\|C - P_i\|\right) ,$$

  for the centroid C of the geometry and all vertices P_i.
- Number of triangles $N_{trimesh}$: Indicates how many triangles are part of the geometry, when the mesh is triangulated.
- Average edge length of triangles $\bar{L}_{edge}$: Measure of the average triangle size. Smaller triangles can lead to increased traversal costs in acceleration structures.
- Maximum edge length of triangles $L_{edge_{max}}$: Indicates the largest triangle in the scene. Larger triangles can create large AABBs or KD-tree nodes, potentially increasing intersection tests.
- Scene volume V_{scene}: Indicates the overall size of the geometry. Larger volumes may require more traversal steps in acceleration structures.
- Average distance between polygon centroids $\bar{d}_{centroids}$: Gives a sense of the scene's spatial distribution. Larger distances might imply that the scene is more sparse, which can affect the efficiency of acceleration structures. For simplicity, the value is sampled using a subset of all centroids.
- Scene depth complexity $D_{complexity}$: Provides an estimate of the number of overlapping primitives in the scene (the depth). Higher depth complexity can lead to slower rendering times due to more ray-primitive intersection tests.

- Aspect ratio $\frac{L_{max}}{L_{min}}$, where L_{max} is the longest dimension of the AABB, and L_{min} is the shortest dimension of the AABB: Indicates the shape of the geometry's bounding box. High aspect ratios may lead to less efficient traversal in acceleration structures due to larger empty spaces.

While we can obtain most characteristics intuitively, the scene depth complexity has to be measured empirically:

$$D_{complexity} = \frac{\sum_{j=1}^{m} \sum_{i=1}^{n} (\text{ray}_j \cap \text{primitive}_i)}{m} . \tag{4.1}$$

For m sample rays we iterate over all n primitives and accumulate the number of intersections. For our case, we generate 1000 sample rays to ensure a reliable approximation of scene depth complexity. This empirical approach allows us to precisely quantify the complexity of ray interactions within the scene.

4.2 Geometries

In this chapter we introduce all geometries in more detail, that have been used for the experimental parts of this thesis. The geometries are part of an internal test set for Molflow development and consists of vacuum chambers from a wide range of studies. We have selected geometries with different properties that describe typical cases sufficiently well. Further, the different properties could reduce potential bias when comparing the results. Here, only non-artificial geometries are mentioned. This is followed by a cylindrical tube which is generated by the application with a set of variable parameters.

ELENA electron gun The electron gun of the ELENA ring, which has already been shown in figure 4.1, served as the prime candidate when investigating the algorithms used in this research study. Although the actual number of facets is considerably low, it has a rather complex interior geometry due to the mesh that represents the NEG (Non-evaporable getter) coating nested inside the outer pump. NEG coatings serve as a pump (as discussed in chapter 2.1) and can be accurately modelled with a sticking coefficient.

FCC concept A section of the FCC (Future Circular Collider) concept design (see figure 4.2) consisting mostly of absorbing and desorbing facets, where repeating parts are connected via teleport facets. This geometry consists of many ($n \geq 4$) polygons and has a comparably large scene depth complexity, where the median for the polygon area for all facets is relatively small.

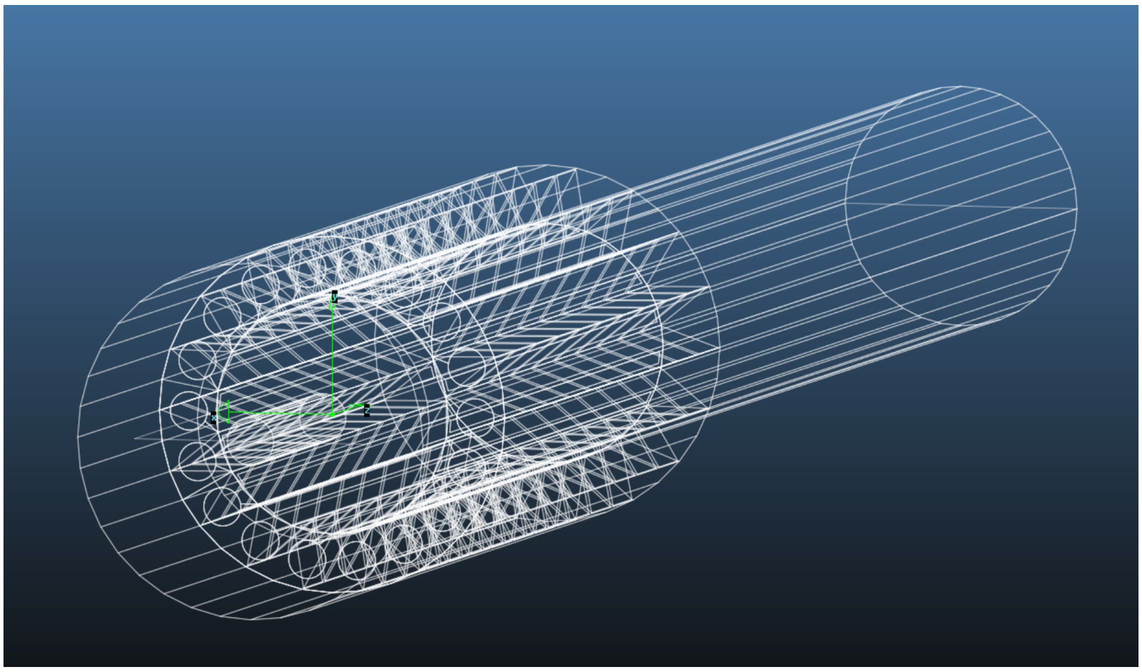


Figure 4.1: Molflow geometry of the electron gun for ELENA ring.

Teleport geometry The accelerator component shown in figure 4.3, also served for the initial study. It also has a low facet count, but a large scene volume and a small primitive density in relation to the average primitive distance.

HEL light cathode Further, we consider the Hollow Electron Lens (HEL) light cathode for the upgrade of the LHC (see figure 4.4). It has a large amount of facets, but is mainly consisting of triangles, which have for the most part small edges. The scene has a large aspect ratio but does not have a very large primitive density or scene depth complexity.

RF cavity The final geometry is shown in figure 4.5. The model represents a super conducting radiofrequency (RF) cavity from an internal design study. It has a large number of facets, mainly triangles, with relatively small areas and short edges. The scene has a large depth complexity but a small primitive density.

Tables 4.1, 4.2, 4.4 and 4.4 show the values for the before mentioned characteristics from chapter 4.1 for the chosen test cases. The test cases differ significantly in most of their properties, which creates an coherent test set for our study.

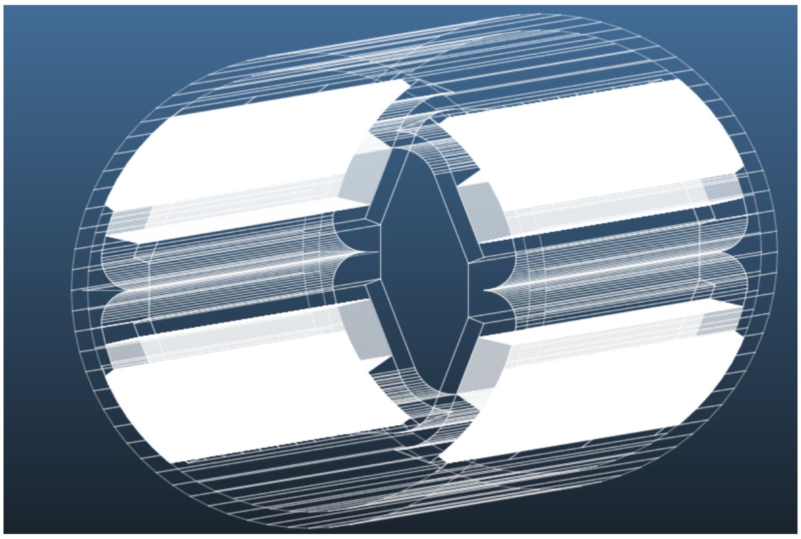


Figure 4.2: Section of a vacuum chamber from CERN's FCC concept (Roberto Kersevan, 2022).

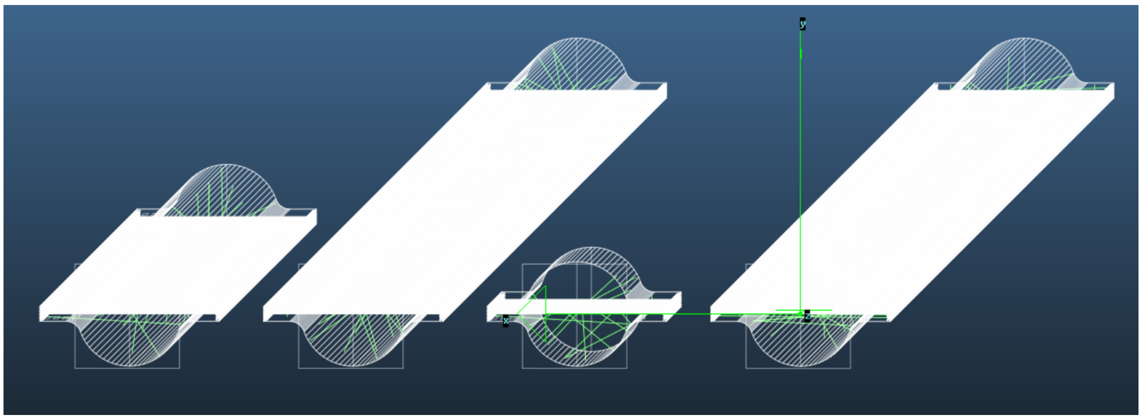


Figure 4.3: Molflow geometry of a vacuum chamber sectioned into four different repeating parts, that are connected via teleport facets.

Table 4.1: Geometry Properties (Part 1)

Filename	N_{fac}	N_{tri}	N_{rect}	$N_{\text{n.poly}}$
ELENA E-Gun	1023	0	955	68
FCCHH Section	140431	0	96379	44052
Teleport section	4678	0	4670	8
HEL light cathode	79159	79159	0	0
RF cavity	90661	76874	13599	188

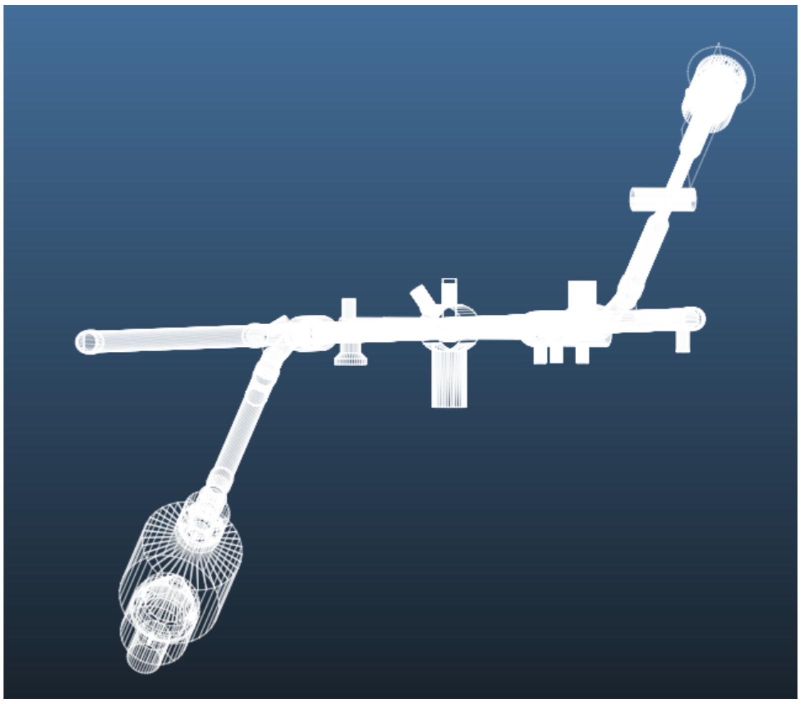


Figure 4.4: Vacuum chamber relying on cold cathode gauges for the valves (Jens, 2021).

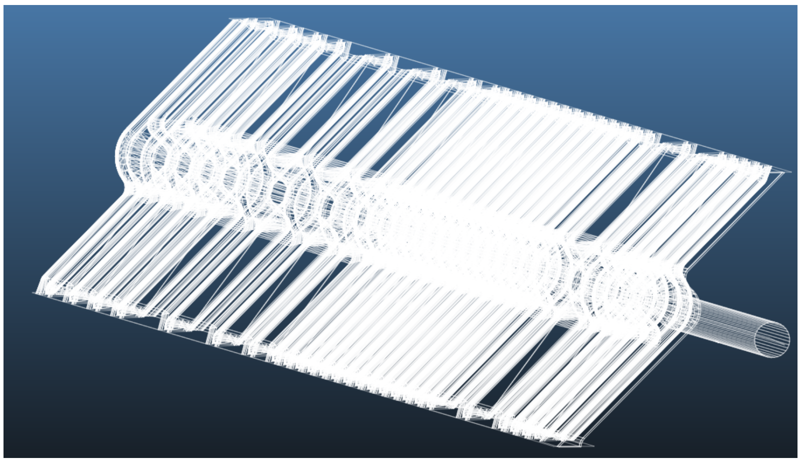


Figure 4.5: Super conducting RF cavity for an internal design study.

Table 4.2: Geometry Properties (Part 2)

Filename	N_{vert}	$\bar{N}_{\text{vert}}$	$N_{\text{med vert}}$	$\tilde{A}_{\text{poly}}$
ELENA E-Gun	3894	7.12708	4	0.671745
FCCHH Section	292478	6.12469	4	0.00224509
Teleport section	17688	4.2018	4	6
HEL light cathode	39575	3	3	0.0282172
RF cavity	55018	3.15834	3	0.00195503

Table 4.3: Geometry Properties (Part 3)

Filename	r_{bound}	$N_{\text{tri mesh}}$	$\bar{L}_{\text{edge}}$	$L_{\text{edge max}}$
ELENA E-Gun	56.9891	5245	3.86231	113.442
FCCHH Section	715.004	579234	32.4358	1430
Teleport section	505.805	10300	158.892	1000.02
HEL light cathode	381.509	79159	2.10945	715.025
RF cavity	65.5495	105016	0.264433	109.336

Table 4.4: Geometry Properties (Part 4)

Filename	V_{scene}	$\bar{d}_{\text{centroids}}$	$D_{\text{complexity}}$	L_{max}/L_{min}
ELENA E-Gun	25087.1	0.209071	1652000	3.02909
FCCHH Section	27684.9	20.9224	266795000	0.565751
Teleport section	402991	0.0255589	1346000	13.596
HEL light cathode	1.10471	0.00716559	37686000	0.594794
RF cavity	6813.11	15.4138	49142000	0.0285556

4.3 Cylindrical vacuum tube

For our study we utilize a cylindrical tube that serves as a vacuum component, for which an analytical solution can be used for the benchmarks (see Gómez-Goñi and Lobo, 2003). A good simulation should lead to a good approximation of this reference solution, making it a perfect candidate for initial validation of an algorithm. The cylindrical inlet (desorption facet) defines a steady influx of particles with an outgassing rate of $10\text{mbar}\cdot l \cdot \text{s}^{-1}$. In addition, both end facets serve as perfect absorbers (sticking coefficient $s = 1$), removing all particles from the system. For the side facets, they are defined as perfectly diffusive – reflecting all particles. The tube can be defined with various length/radius ratios (L/R). In Molflow the circular form is approximated with a finite amount of side facets. In our numerical experiments, the L/R ratios and the levels of approximation were varied, as indicated in the respective sections. Further, utilizing cylindrical tubes with varying parameters is ideal as they reduce a vacuum problem to a geometrically simple test case.

Figure 4.6 shows the geometry in Molflow with an L/R ratio of 10. In this example, the circular inlet (left) and outlet (right) are each approximated with 100 vertices, which results in 100 rectangular facets for the side walls, thus 102 polygons in total.

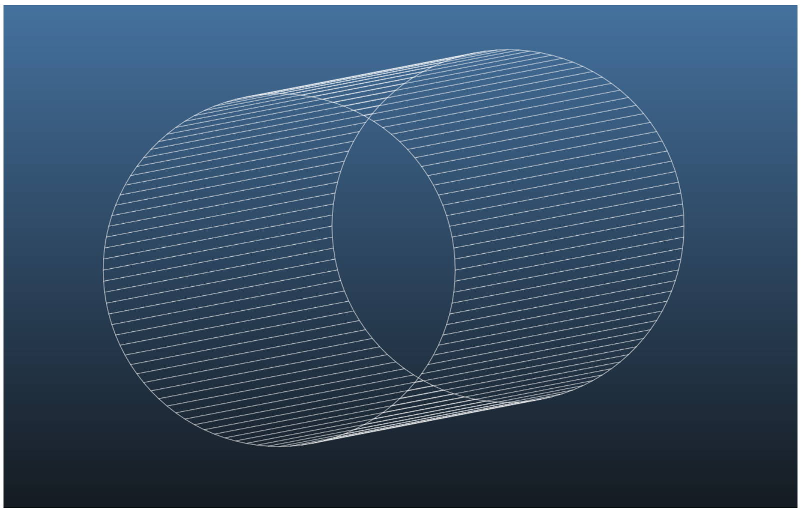


Figure 4.6: Cylindrical tube in Molflow. The left side serves as an inlet, the right side serves as an outlet. Approximation level is given the amount of side facets. More facets lead to a better approximation of the circular shape.

5 Design and Development

Various components of Molflow and Synrad that have been developed or optimised as part of this work are elaborated on in this chapter by explaining the methodology that has been used as well as their final design.

5.1 Software architecture

For the general structure, some components must be accessible independently from the GUI application Molflow and the CLI application MolflowCLI, which is particularly interesting for the Simulation components. The applications are therefore sharing a common SimulationManager that is handling the coordination with one of the selected SimulationCores. The proposed modules are visualised in more detail in figure 3.1.

5.2 Geometry representation

From a design perspective, we consider Molflow to serve two main purposes: The CAD-like graphical interface, where vacuum components can be modelled, and the vacuum simulations. There are various methods for representing geometries. Commonly polygon or triangle meshes are used for efficient representation. Polygon meshes are the preferred representation for geometries in Molflow's CPU implementation. Triangle meshes are used for the developed GPU kernel (see chapter 8). For completeness, we discuss another alternative method for representation with CSG.

5.2.1 Meshes

Meshes, serving as tessellated surfaces, are widely utilized for geometric representation, especially in Computer-Aided Design (CAD) applications. They decompose geometries by creating a network of interconnected, simpler polygonal elements. This makes them highly adaptive and usable to model very complex geometries. With a finer approximation, the memory requirements also increase. In Molflow we utilize polygon meshes to represent all models. Furthermore, in the context of GPU simulations (elaborated in chapter 8), we implemented the representation with a triangle mesh.

Polygon mesh

General polygons have the advantage that they create a common interface for all types of primitives. CAD geometries, which are typically given as a triangle mesh, can be explicitly used as a polygon mesh. Further, Molflow allows to merge triangles into polygons when they lie more or less on the same plane. Using this approach, a polygon mesh could significantly reduce the number of vertices and facets in a model. While these simplifications might impact the accuracy minimally, this reduces the memory requirements as less information has to be saved, e.g hit counters per facet (see chapter 2.2.3).

Polygon meshes can be described in many ways, in Molflow's C++ interface they are represented as follows. A mesh contains an array of polygons and an array of 3d vertex coordinates. An individual `polygon` contains an array with vertex indices. With the indices the corresponding vertices forming the polygon can be fetched.

The data structures describing a polygon mesh and an individual polygon object are sketched in figure 5.1. A polygon mesh is defined by two vectors. A polygon vector contains all N polygons P of the geometry and a vector containing all N_V vertices V_j with 3D world coordinates, denoted (x_j, y_j, z_j). Each polygon is defined by a vector containing the precomputed local 2D coordinates of the polygon (see section 2.3.2). The local 2d coordinates are used for simplified ray tracing calculations or texture lookups. Also, each polygon maintains a vector of indices I_i used to directly access the corresponding 3D vertex. Like this, vertices that are contained in multiple polygons have to be stored only once. The data structure can be used for polygons with varying amounts of indices. For each polygon, the order of the vertices describes the orientation of the facet normal according to the left-hand rule.

Triangle mesh

In GPU applications, such as the developed GPU kernel for Molflow (see chapter 8), triangle meshes offer significant advantages, primarily due to their uniform structure and compatibility with the SIMD (Single Instruction Multiple Data) architecture of GPUs. By exclusively using triangles, the simplest and most consistent polygon type, we can create a highly optimized data structure that aligns well with GPU processing capabilities. This uniformity benefits both the intersection algorithms used in ray tracing, and for the efficient data representation, greatly enhancing computational efficiency and performance in GPU-based simulations. Typically, an efficient triangle mesh is implemented in a Structure of Arrays (SoA) data model. In this model, the representation of a triangle is implicit. The data structure consists of three arrays:

1. Index array: This array contiguously stores index triplets corresponding to each triangle's vertices in ascending order, based on the triangle ID.

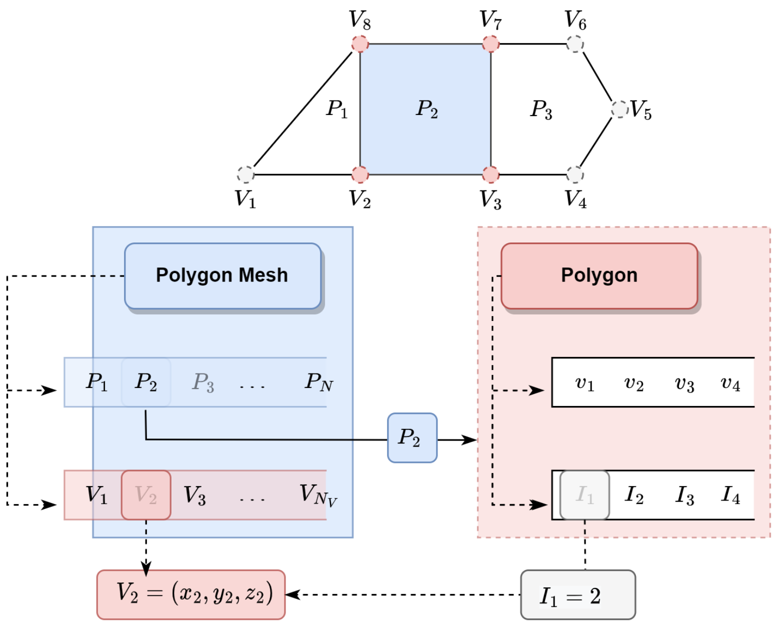


Figure 5.1: Sketch of Molflow's data structure to store a polygon mesh and polygon primitives. A polygon mesh stores instances of Polygons P and 3d vector coordinates $\mathbf{V}$. A polygon consists of local 2d coordinates $\mathbf{v}$ and indices $I_i = k$ referencing the real vector coordinates V_k. In the example, the polygon P_2 is accessed. Its first vertex pointer $I_1 = 2$ is pointing to the global vertex $\mathbf{V}_2$. In this case, P_2 is a $N = 4$-polygon, so it consists of four 2d vertices and four pointers to vertex indices.

2. Vertex coordinates array: Another array holds the 3D world coordinates $\mathbf{V} = (x, y, z)$ of the vertices.

3. Local 2D coordinates array: The third array contains triplets of local 2D coordinates for each triangle, also ordered by the triangle ID.

The size of the arrays holding the indices and local 2D coordinates is $3N$, where N is the total number of triangles. The vertex coordinates array size is N_V, representing the total number of unique vertices in the geometry. This approach effectively eliminates redundancy by avoiding the storage of duplicate vertex data.

For a given triangle with an index T in the range $[0, N-1]$, its corresponding vertex indices can be calculated using a stride ($s = 3$). For instance, for triangle $T = 1$, the index triplet begins at $I_{start} = T \cdot s = 3$ and extends to $I_{end} = (T+1) \cdot s - 1 = 5$. Each index I then maps to a corresponding vertex $\mathbf{V}$. Similarly, the local 2D coordinates are accessed using the same approach using the triplet spanning array positions $[I_{start}, I_{end}]$. We sketch an example and the mesh representation for an arbitrary geometry in figure 5.2.

5.2.2 Constructive solid geometry

In addition to meshes,Constructive solid geometry (CSG) is another method of representing geometries that merits consideration, especially in the context of applications like Molflow. CSG uses a series of Boolean operations to combine basic geometric shapes into more complex forms. This method is known for its precision and efficient use of memory, making it an appealing choice for certain types of computational geometry tasks. CSG operates on volumes rather than surfaces, allowing for the construction of highly detailed and precise geometric models. This can be particularly advantageous in simulations where accuracy is important. Since CSG represents objects using a combination of simple shapes and Boolean operators, it can be more memory-efficient than mesh representations, especially for geometries that can be easily broken down into basic volumetric shapes.

However, despite these advantages, CSG is not the preferred method for Molflow. This decision is rooted in the specific requirements of vacuum simulations and CAD geometry handling in Molflow. The primary focus of Molflow is on efficient ray tracing and handling complex CAD-generated geometries, which align better with tessellated surface representations like polygon and triangle meshes. Additionally, texture mapping and certain computational aspects, crucial for Molflow's functionality, are more straightforward with mesh geometries compared to CSG. Thus, while CSG offers significant benefits in precision and memory efficiency, its application in Molflow is limited due to the software's specific functional and performance requirements.

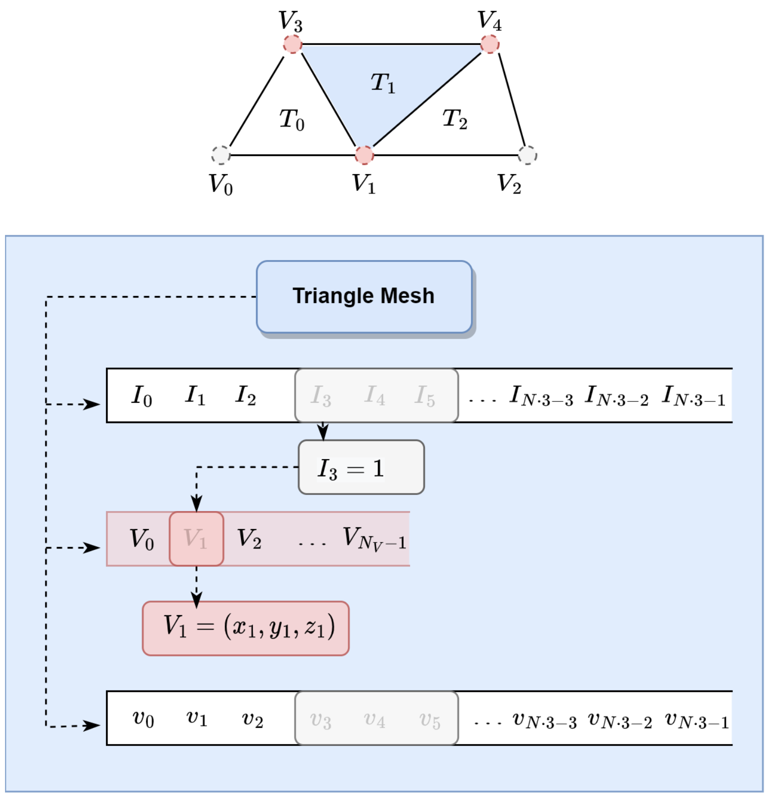

V_3
V_4
T_1
T_0
T_2
V_0
V_1
V_2
Triangle Mesh
I_0 I_1 I_2 I_3 I_4 I_5 $\dots$ $I_{N\cdot3-3}$ $I_{N\cdot3-2}$ $I_{N\cdot3-1}$
$I_3 = 1$
V_0 V_1 V_2 $\dots$ V_{N_V-1}
$V_1 = (x_1, y_1, z_1)$
v_0 v_1 v_2 v_3 v_4 v_5 $\dots$ $v_{N\cdot3-3}$ $v_{N\cdot3-2}$ $v_{N\cdot3-1}$

Figure 5.2: Sketch of the SoA data structure for a triangle mesh. The example shows triangle $T = 1$ being accessed through its vertex index triplet $\{I_3, I_4, I_5\}$. The location of the triplet is computed using the triangle ID. The first index I_3 in this example points to the global vertex $\mathbf{V}_1$ obtained from the structure stored with the mesh. Additionally, the corresponding local 2D coordinates are accessed using the same computed triplet location.

There are a few simulation tools for physical problems that deploy CSG models such as OpenMC (see Romano et al., 2015). OpenMC is a Monte Carlo particle transport simulation code commonly used for modeling of nuclear reactors, which is capable of simulating various nuclear reactions for neutrons and photons. They also adapted their CSG ray tracer to make use of RTX hardware-acceleration via OptiX using tree-traversal on top of a custom intersection test for CSG geometries. Furthermore, they compared its performance to a triangle mesh making full use of hardware-acceleration (see Salmon and McIntosh-Smith, 2019). In general they achieved a better performance when using triangle meshes, where "On RT core accelerated triangle meshes the speedups [are around] $\sim$ 33x [and] $\sim$ 10x on CSG models."

As CSG geometries are rather unknown among engineers creating models or running simulations for vacuum problems, the use of tessellated meshes is a natural choice. Meshes, when compared to CSG geometries, benefit further from being highly optimized in the context of ray tracing. The use of meshes aligns well with the need for efficient computation and effective texture mapping, making them particularly suitable for Molflow.

5.3 Time-dependent simulations

Time-dependent simulations allows users to analyze and understand vacuum systems that are not in a steady-state. They add an additional overhead to the simulations as statistics need to be gathered not only spatially on facets, but also temporal.

This chapter gives a brief explanation of the difference between steady-state and time-dependent simulations. Further, we elaborate the usage of so-called time moments that were initially implemented by Marton Ady (2016). We elaborate on our modification for the original algorithm with additional constraints and an optimized lookup method. The underlying difficulties in terms of developing a performant and memory-efficient simulator are also discussed. To evaluate the efficiency of time-dependent simulations in both time and space and to further validate the correctness of the simulations, a case study of the CLIC design study (Burrows et al., 2018) is used.

5.3.1 Motivation

In Molflow it is possible to simulate in the time domain (see chapter 2.2.4) by tracking the particle speed. A successful query then returns the hit facet, the particular hit location, and also the time of the hit. This can be interesting not only to understand and visualise the distribution of when particles reach certain parts of the system but also when time-dependent system parameters are applied. Certain system parameters such as a facet's outgassing rate or sticking factor can vary in time. They can be set as as time-dependent by giving a set of time-value points. Corresponding to a specific time point, the time-dependent system parameter can then be obtained by using linear interpolation between the user-defined values and by constant extrapolation outside of these. As elaborated in chapter 2.2.4, physical quantities can be translated from steady-state to time-dependent. Marton Ady (2016) implemented this using so-called *time moments* by utilizing counting bins, that correspond to a specified time window size. During a simulation, a Monte Carlo event is then accumulated in the bin that corresponds to the specific time. In this work, we reevaluate the technique developed, as it has a major impact on performance, as previously shown in the benchmark shown in table 3.1. Further, we develop and implement a new version by defining a constraint allowing us to leverage better lookup algorithms.

5.3.2 Time moments

In Molflow, users are interested in how a system behaves at particular points in time. They can define one or several series of so-called *time moments* for points in time $t_1, t_2, ..., t_N$ for which the simulation will evaluate the simulation results as a function of time in addition to the steady-state results. A time moment spans an interval, that refers to a specific point in time and a corresponding time window. Numerically, it is necessary to gather statistics in an interval to increase the sample size. Thus, for each point in time t_i we create a time bin that covers the range $t_i - t_w/2$ to $t_i + t_w/2$ for a time window t_w. A series of time moments is given by a starting time t_{start} and an end time t_{end}, which is then divided based on a fixed *time step size* $\mathrm{d}t$. This will result in a series of $N = 1 + \lfloor (t_{end} - t_{start})/\mathrm{d}t \rfloor$ time moments, where $t_i = (i-1) \cdot \mathrm{d}t + t_{start}$, so $t_1 = t_{start}$ and $t_{end} = (N-1) \cdot \mathrm{d}t + t_{start}$. To avoid overlaps between individual bins, constraint $t_w \leq \mathrm{d}t$ is imposed. Thus, a user can define this Specified Time Series (STS) in the form of a tuple $T = \{t_{start}, t_{end}, t_w, \mathrm{d}t\}$. In figure 5.3 two STS T_1 and T_2 are sketched with their individual parameters.

5.3.3 Redesign

The initial design for the time-dependent simulations introduced by Marton Ady (2016) greatly increased the extent of possible use cases. Monte Carlo events, in the sense of collisions, could only be tracked by a series of time points with a fixed global time window $t_{w,glob}$, giving the tuple $\{t_{start}, t_{end}, \mathrm{d}t\}$. The flexibility to individually adjust the time window for each STS and a necessary improvement to the increased computational demands of time-dependent simulations required a redesign of the corresponding data structures and algorithms. As it has been found, the lookup function to find the bin index of the result data structure for the corresponding

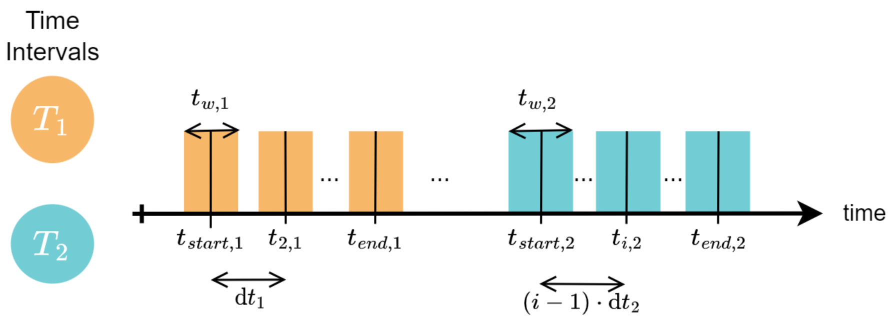


Figure 5.3: The bins of two time series T_1 and T_2 are sketched. Here, we distinguish the individual parameters of each time series T_m with a secondary index: $T_m = \{t_{start,m}, t_{end,m}, t_{w,m}, \mathrm{d}t_m\}$

event at time t had the biggest impact on the simulation performance when using time-dependent statistics.

With the approach previously used, for a given time point t all time moments of the STS had to be checked. It was straightforward to see, that better lookup algorithms could be defined by imposing the constraint that time bins are strictly non-overlapping. It is then possible to sort the non-overlapping time intervals beforehand. This allows the use of algorithms such as binary search reducing the time complexity for the search of a specific time point from $\mathcal{O}(T)$ to $\mathcal{O}(\log(T))$, where T is the number of moments. In the following section we investigate a set of algorithms that can be used for the index lookup routine.

5.3.4 Lookup algorithms

Most simply, the series of moments and their time windows can be given as an ordered array of pairs $(t_i - \frac{t_w}{2}, t_i + \frac{t_w}{2})$ for which conventional search algorithms can be deployed. In addition, these algorithms can make use of the fact that for time-dependent problems usually a series of monotonically increasing time moments is queried. In Molflow, as a simulated particle will only be simulated forward in time, a particle's time t_p will always increase with each event until it is removed from the system ($t_p < t_{p+1}$ for each particle state p). This property can be accounted for to reduce the search space e.g. with a starting index.

Where a binary search seems to be a natural first choice, we decided to investigate the usability of other search algorithms that seem more suitable for the underlying problem (time advancement), namely Interpolation Search and Jump Search. Further, we also propose a different solution that is not based on finding the solution for a search problem, here the bin indices are calculated directly.

Interpolation Search Interpolation Search (Perl, Itai, and Avni, 1978) is an ideal candidate for this problem due to its reduced average complexity for uniformly distributed values which is $O(\log\log \mathrm{N})$. Time-dependent simulations in Molflow usually have a search space consisting of one or many individually uniformly distributed time moments. The algorithm works on the assumption, that for sorted uniformly-distributed values a queried key will be close to an interpolated value that serves as a starting point.

Given the query x, an array of sorted values A, a lower bound l and an upper bound h a starting position p can be calculated as

$$p = l + \frac{(x - A_l) \cdot (h - l)}{A_h - A_l} . \tag{5.1}$$

Here, the indices to A_i denote the element at the corresponding position i. With the initial bounds $l = 0, h = N - 1$, where N is the amount of values in A, each search iteration will change either the lower bound in the case $A_p < x$ to $l = p + 1$ or the upper bound in the case of $A_p > x$ to $h = p - 1$.

Jump Search Jump Search (Shneiderman, 1978) does not have the best average time complexity with $O(\sqrt{N})$, but has another property that makes it an interesting choice for Molflow's time-dependent simulations: it is progressing with a fixed step size s. Similar to Interpolation Search, this can be favourable for uniformly distributed values. The algorithm goes as follow:

1. Determine the step size s, typically set to $\sqrt{N}$.
2. Initialize a variable l to 0 to keep track of the previous step's last index.
3. Start a loop until the target value is found or determined to be absent:
   a) Calculate the start index of the current block $h = \min(l + s, N - 1)$.
   b) Perform a linear search within the block defined by indices l to h.
   c) If the target value x is found, return the index of the matching element.
   d) If the current element is greater than x, return to the previous block.
   e) If the end of the array is reached without finding x and $l \geq N$, the target value is not present.

In theory, jump search has favourable traits when values are uniformly distributed and the number of jumps can be minimised. In Molflow, we can influence the latter by using a starting index to initialize the lower bound l. In ideal scenarios, where $s \approx \mathrm{d}t_{avg}$, for the average time step size $\mathrm{d}t_{avg}$, this can reduce the search space almost completely to the part applying linear search.

Calculative binning

A more direct approach can be found by directly computing the bin index given to all STS. Instead of keeping the individual time bins $t_i - \frac{t_w}{2}, t_i + \frac{t_w}{2}$ in memory, for every STS a starting index $\sigma_{i,off}$ is saved. For the first time series, the offset is $i = 0\,, \quad \sigma_{0,off} = 0$ and for the i^{th} it is $\sigma_{i,off} = \sigma_{i-1,off} + N_{i-1}$, where N_{i-1} is the amount of individual time moments in the $(i-1)^{th}$ time series.

First, to find the bin index k for a simulation time point t, the STS must be determined for which $t_{start} \leq t \leq t_{end}$. For a match the local key k_{loc} can then be calculated with:

$$k_{loc} = \lfloor \frac{t - t_{start} + 0.5 \cdot t_w}{\mathrm{d}t} \rfloor\,. \tag{5.2}$$

Only if

$$t > t_{start} + k_{loc} \cdot \mathrm{d}t - 0.5 \cdot t_w \quad \wedge \quad t < t_{start} + k_{loc} \cdot \mathrm{d}t + 0.5 \cdot t_w \tag{5.3}$$

then the time point t is matched into the bin. The index for the global structure k_{glob} is then simply $k_{glob} = k_{loc} + \sigma_{i,off}$.

All operations can be solved in O(1) to get the bin index, where only finding the matching STS is $\mathrm{O}(\log S)$, where S is the amount of STS. However, in practical applications S is usually very low $S \leq 10$, making the calculation of the bin index potentially the ideal choice.

5.3.5 Benchmarks

This chapter discusses the benchmarks performed on the proposed algorithms for time-dependent calculations. First, the old algorithm was improved by replacing it with binary search. However, to account for all possible scenarios, other algorithms have been implemented. Finally, a set of test cases with varying characteristics for the partitioning in time was created to conclude the algorithmic benchmarking.

A basic implementation was achieved with a simple vector structure. The non-overlapping intervals have to be sorted, which allows the use of search techniques such as binary search. Coupled with a better binary search and a new constraint to avoid non-overlapping intervals, the simulation for a simple pipe geometry with 5 side facets (see chapter 4.3) and 1000 time windows was already accelerated by a speed-up factor of about 9× for the whole simulation. The benchmark values are shown in table 5.1 for 3 independent test runs. This test geometry was chosen to highlight the impact of the algorithms that are used in addition for time-dependent simulations.

To put an emphasis on the lookup algorithm, we created a test suite TDBench for which a set of the before mentioned algorithms have been implemented. The

	Run #1 [in s]	Run #2 [in s]	Run #3 [in s]
Old	26:42	26:11	26:09
New	3:00	3:04	3:01

Table 5.1: Simulations using the original algorithm in Molflow *Old* and a first implementation using binary search *New*. Input geometry is a pipe geometry with 5 side facets and $N = 1000$ time windows. Time is given for the full simulation in 3 independent simulation runs for the same number of desorbed particles.

algorithms can work directly on a set of test cases without any overhead from the actual Molflow routines. In TDBench the test cases are pre-computed or predefined. Implemented are the following algorithms:

- BS $\leftarrow$ Binary search
- IS $\leftarrow$ Interpolation search
- JS $\leftarrow$ Jump search
- CB $\leftarrow$ Calculative binning

Furthermore, some of the algorithms are implemented with the capability of utilising a starting index to continue from the previous location, as elaborated previously in the chapter 5.3.3. This supports the idea that an existing particle advances in time until it is removed from the system, and thus time moments prior to the last time point that is giving the starting index can be skipped. The capability has been implemented for Binary search, Jump search and Interpolation search.

TDBench has the option to run on an artificial test case or on an extract from a real Molflow simulation. Test cases consist of sequences $p_m = \{t_{m,0}, \ldots, t_{m,\text{LAST}}\}$ corresponding to individual particles m. Each sequence contains $\text{LAST} - 1$ time points $t_{m,k}$ for the first event $k = 0$ up to the last event $k = \text{LAST}$, the particle absorption, where $\text{LAST} > 0$. The sequence length LAST is individual for each particle. A benchmark for a given algorithm and test case works by iterating over all time sequences of the test case and incrementing the bin with the index that has been looked up. For algorithms making use of a start index, this index is set to the looked up index and reset to 0 when a new time sequence is looped over.

Artificial test cases

First, we compare the algorithms against a set of artificial test cases of differing intervals. Here, the input is artificially generated and is not based on a real Molflow test case. To emulate some of the behaviour from a real Molflow simulation, the input is generated by the following principle. The generator is given a tuple $\text{TP} = \{n, t_{min}, t_{max}, \Delta t_{min}, \Delta t_{max}\}$ describing the time points. Here, n is the number of time moments to be generated, t_{min} and t_{max} are the smallest and largest points in time that can be generated, respectively. Δt_{min} and Δt_{max} represent the smallest and largest time steps, respectively. With this tuple, sequences in analogy to Molflow's particle tracking are generated. Given some random numbers $\mathsf{r} \in [\Delta t_{min}, \Delta t_{max}]$, a test sequence will start at $t_0 = t_{min} + \mathsf{r}$ and consecutive elements $t_{i+1} = t_i + \mathsf{r}$, $i \geq 0$ are generated. The system generates a test sequence only when $t_i \leq t_{max}$. For the test bench, we use the tuple $\text{TP} = \{10^7, 0.0, 100.0, 0.001, 10.0\}$ to generate the sequences.

The test bench contains a mixed set of artificial test cases, which should highlight particular strengths and weaknesses of the proposed algorithms. E.g. interpolation search and jump search are expected to perform well on uniformly distributed data sets, but not so much for others. We define the following set of artificial test cases, where each test case contains at least one STS:

1. $T_0 = \{0.001, 100.0, 0.1, 0.1\}$
   This test case has $N \approx 1,000$ equidistant time points. No gaps between time points, as $t_w \equiv \mathrm{d}t$.

2. $T_0 = \{0.001, 100.0, 0.01, 0.01\}$
   This test case has $N \approx 10,000$ equidistant time points. No gaps between time points, as $t_w \equiv \mathrm{d}t$.

3. $T_0 = \{0.001, 100.0, 0.001, 0.001\}$
   This test case has $N \approx 100,000$ equidistant time points. No gaps between time points, as $t_w \equiv \mathrm{d}t$.

4. $T_0 = \{0.001, 100.0, 10^{-7}, 0.1\}$
   This test case has $N \approx 1,000$ equidistant time points. And large gaps between each data point. Bins cover $\sim 10^{-4}\%$ of the whole interval.

5. $T_0 = \{0.001, 100.0, 10^{-7}, 0.001\}$
   This test case has $N \approx 100,000$ uniformly distributed time points. And gaps between each data point. Bins cover $\sim 10^{-2}\%$ of the whole interval.

6. $T_0 = \{0.001, 1.0, 0.001, 0.001\}$,
   $T_1 = \{1.01, 10.0, 0.001, 0.001\}$,
   $T_2 = \{10.01, 20.0, 0.001, 0.001\}$,
   $T_3 = \{20.01, 30.0, 0.001, 0.001\}$,
   $T_4 = \{30.01, 40.0, 0.001, 0.001\}$,
   $T_5 = \{40.01, 50.0, 0.001, 0.001\}$,

$T_6 = \{50.01, 60.0, 0.001, 0.001\}$,
$T_7 = \{60.01, 70.0, 0.001, 0.001\}$,
$T_8 = \{70.01, 80.0, 0.001, 0.001\}$,
$T_9 = \{80.01, 90.0, 0.001, 0.001\}$,
$T_{10} = \{90.01, 100.0, 0.001, 0.001\}$
This test case has $N \approx 100,000$ uniformly distributed time points across multiple STS. No gaps between data points of each series. Only gaps between each time series.

7. $T_0 = \{0.001, 1.0, 0.01, 0.01\}$,
$T_1 = \{1.01, 10.0, 0.01, 0.01\}$,
$T_2 = \{10.01, 20.0, 0.01, 0.01\}$,
$T_3 = \{20.01, 30.0, 0.01, 0.01\}$,
$T_4 = \{30.01, 40.0, 0.01, 0.01\}$,
$T_5 = \{40.01, 50.0, 0.01, 0.01\}$,
$T_6 = \{50.01, 60.0, 0.01, 0.01\}$,
$T_7 = \{60.01, 70.0, 0.01, 0.01\}$,
$T_8 = \{70.01, 80.0, 0.01, 0.01\}$,
$T_9 = \{80.01, 90.0, 0.01, 0.01\}$,
$T_{10} = \{90.01, 100.0, 0.01, 0.01\}$
This test case has $N \approx 10,000$ uniformly distributed time points across multiple STS. No gaps between data points of each series. Only gaps between each time series.

8. $T_0 = \{0.001, 1.0, 0.1, 0.01\}$,
$T_1 = \{1.01, 10.0, 0.1, 0.01\}$,
$T_2 = \{10.01, 20.0, 0.1, 0.01\}$,
$T_3 = \{20.01, 30.0, 0.1, 0.01\}$,
$T_4 = \{30.01, 40.0, 0.1, 0.01\}$,
$T_5 = \{40.01, 50.0, 0.1, 0.01\}$,
$T_6 = \{50.01, 60.0, 0.1, 0.01\}$,
$T_7 = \{60.01, 70.0, 0.1, 0.01\}$,
$T_8 = \{70.01, 80.0, 0.1, 0.01\}$,
$T_9 = \{80.01, 90.0, 0.1, 0.01\}$,
$T_{10} = \{90.01, 100.0, 0.1, 0.01\}$
This test case has $N \approx 1,000$ uniformly distributed time points in multiple STS. Gaps between data points of each series and between each time series.

9. $T_0 = \{0.001, 1.0, 0.1, 0.01\}$,
$T_1 = \{1.1, 10.0, 0.001, 0.001\}$,
$T_2 = \{10.01, 20.0, 0.1, 0.00001\}$,
$T_3 = \{20.01, 30.0, 0.1, 0.0001\}$,

$T_4 = \{30.01, 40.0, 0.1, 0.001\}$,
$T_5 = \{40.01, 50.0, 0.00001, 0.00001\}$,
$T_6 = \{50.01, 60.0, 0.05, 0.001\}$,
$T_7 = \{60.01, 70.0, 0.1, 0.01\}$,
$T_8 = \{70.01, 80.0, 0.7, 0.07\}$,
$T_9 = \{80.01, 90.0, 0.1, 0.01\}$,
$T_{10} = \{90.01, 100.0, 0.09, 0.001\}$
This test case has $N \approx 1,000,000$ time points that are mostly concentrated in STS (T_5). Irregularly distributed data points with varying gap sizes between data points of each time series.

10. $T_0 = \{0.001, 1.0, 0.2, 0.01\}$,
$T_1 = \{1.7, 10.0, 1.0, 0.01\}$,
$T_2 = \{10.7, 20.0, 1.0, 0.01\}$
This test case has $N = 24$ time points. Gaps exist between data points and time series. Time series do not cover the entire test interval $[0.001, 100.0]$.

Test cases #1, #2 and #3 will show the effect of varying amount of bins for otherwise similar data sets. Test cases #4 and #5 also highlight varying amounts of bins, where there are gaps between each bin. Test cases #6 and #7 also differ in the amount of bins, highlighting the effect of multiple time series compared to the first three test cases. Test case #8 is similar to #6 and #7 but between each bin are gaps. Test case #9 highlights the effect of multiple time series with varying amounts of bins, one time series is accounting for most time bins in total. Test case #10 focuses on a small set of time series and a minimal amount of data points, where around 70% of time events will happen outside of the time series range.

In table 5.2 we give the average time over 5 runs for each algorithm and each test case. For each run, a new sequence of time events is generated using the same generator TP as previously described, generating 10^7 time points for the input. The same sequences are used for all test cases and algorithms in a single run. The post fix *wI* (with index) means, the algorithm has been using a starting index to search within a smaller subset of the whole interval. Each algorithm received the same randomly generated input set for each test.

Interpolation search and Calculative binning work best on almost all test cases. Surprisingly, direct calculation of the bin index is not always the best option. In 6 of 10 cases, Interpolation search has the best runtime, where Calculative binning has the best runtime for the remaining test cases. There is an exception, though, for Interpolation search for the test case #9. This test case is conflicting with the assumption that the algorithm works on a uniform distribution of time moments. Only one STS is contributing to the biggest amount of time moments, while it does not span a comparingly large time frame. This leads to the problem, that the initial guess for the search location will be over- or

Alg/TC	BS	IS	JS	CB	BSwI	JSwI	ISwI
#1	497.04	110.97	517.30	**95.03**	388.86	341.60	134.22
#2	704.63	130.85	1136.69	**97.08**	621.10	561.31	165.41
#3	964.57	146.73	3649.91	**96.18**	1023.34	1270.02	233.79
#4	477.87	**50.58**	513.97	87.57	501.58	544.83	60.99
#5	948.87	**65.65**	3428.77	84.35	964.31	3639.04	85.76
#6	984.40	**294.59**	3685.42	337.87	1108.39	1522.05	502.29
#7	685.74	**235.51**	1127.76	339.71	623.14	552.22	351.36
#8	496.58	**68.07**	522.96	346.91	493.63	473.22	79.59
#9	854.53	20943.24	12515.32	**402.48**	1052.77	7715.05	20516.16
#10	78.43	**48.50**	138.70	127.66	92.88	155.42	67.02

Table 5.2: Benchmark of multiple algorithms (*Alg*) to determine appropriate bins for time-dependent data structures and various artificial test cases. The results of the benchmark are given in milliseconds. The best results are highlighted in bold for each test case (*TC*).

undershot by a lot. Essentially leading to the algorithms worst case performance of $O(n)$. Direct calculation is bound to the amount of STS in the set, which will never be considerably large in real cases to have a bad impact. The algorithms using a starting index do not always profit from this capability. Binary search and Interpolation search work great as they can quickly search through the problem space. Jump search on the other hand has larger speed-ups on average as it searches with a static step size.

Real test cases

Further, we compare the algorithms on a real test case, a pump that is part of the CLIC (Compact LInear Collider) design concept (Burrows et al., 2018). This geometry is exclusive to this chapter due to its time dependent properties, which we neglect for other studies focussing only on geometrical properties as elaborated in chapter 4. Here, instead of using artificially generated time sequences, time sequences for $n = 10^7$ time values are gathered from a real simulation. Similar to the artificial sequence, the sampled sequence accounts for one particle at a time, so that algorithms making use of a cached start index can still be used.

For the geometry of the pump, depicted in figure 5.4, we use the following sets of STS as test cases:

1. $T_0 = \{1.0 \times 10^{-6}, 9.1 \times 10^{-6}, 1.0 \times 10^{-6}, 1.0 \times 10^{-6}\}$,
   $T_1 = \{1.0 \times 10^{-5}, 9.1 \times 10^{-5}, 1.0 \times 10^{-6}, 1.0 \times 10^{-6}\}$,
   $T_2 = \{1.0 \times 10^{-4}, 9.1 \times 10^{-4}, 1.0 \times 10^{-5}, 1.0 \times 10^{-6}\}$,

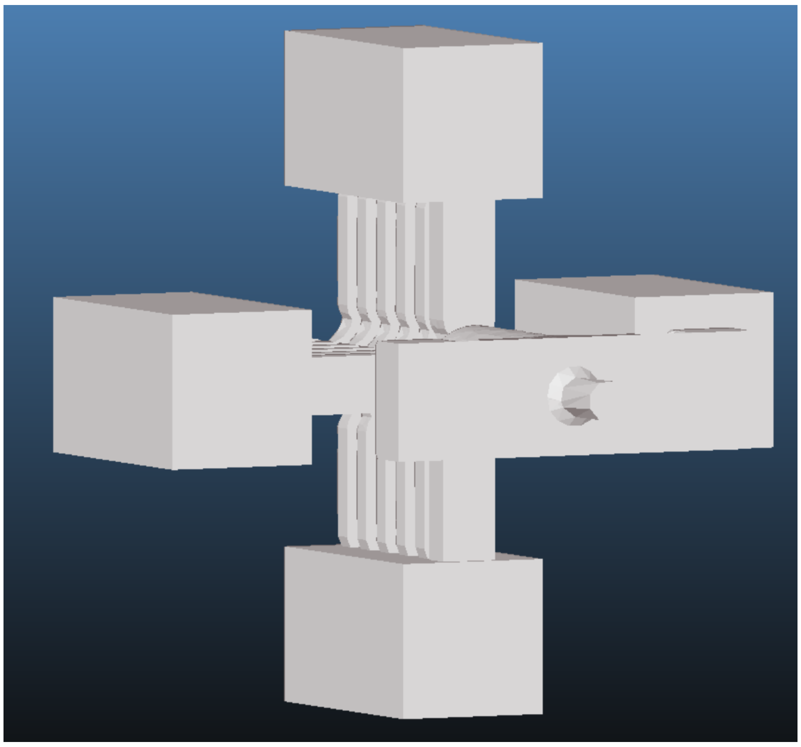


Figure 5.4: A geometry loaded in Molflow for a pump from the CLIC design concept.

$T_3 = \{1.0 \times 10^{-3}, 9.1 \times 10^{-3}, 1.0 \times 10^{-4}, 1.0 \times 10^{-6}\}$
The original set of STS with around $N \approx 300$ time bins. Same time window size for all STS.

2. $T_0 = \{1.0 \times 10^{-6}, 9.9 \times 10^{-6}, 1.0 \times 10^{-11}, 1.0 \times 10^{-11}\}$,
$T_1 = \{1.0 \times 10^{-5}, 9.9 \times 10^{-5}, 1.0 \times 10^{-10}, 1.0 \times 10^{-10}\}$,
$T_2 = \{1.0 \times 10^{-4}, 9.9 \times 10^{-4}, 1.0 \times 10^{-9}, 1.0 \times 10^{-9}\}$,
$T_3 = \{1.0 \times 10^{-3}, 9.9 \times 10^{-3}, 1.0 \times 10^{-8}, 1.0 \times 10^{-8}\}$
The original set of STS with around $N \approx 2,400,000$ time bins. Same time window size for all STS.

3. $T_0 = \{1.0 \times 10^{-6}, 9.9 \times 10^{-6}, 1.0 \times 10^{-11}, 1.0 \times 10^{-11}\}$,
$T_1 = \{1.0 \times 10^{-5}, 9.9 \times 10^{-5}, 1.0 \times 10^{-10}, 1.0 \times 10^{-10}\}$,
$T_2 = \{1.0 \times 10^{-4}, 9.9 \times 10^{-4}, 1.0 \times 10^{-9}, 1.0 \times 10^{-9}\}$,
$T_3 = \{1.0 \times 10^{-3}, 9.9 \times 10^{-3}, 1.0 \times 10^{-8}, 1.0 \times 10^{-8}\}$
The same intervals with a larger amount of time bins $N \approx 3,200,000$. Different time windows, no gaps in data points.

4. $T_0 = \{1.0 \times 10^{-6}, 9.9 \times 10^{-6}, 1.0 \times 10^{-11}, 1.0 \times 10^{-13}\}$,
$T_1 = \{1.0 \times 10^{-5}, 9.9 \times 10^{-5}, 1.0 \times 10^{-10}, 1.0 \times 10^{-12}\}$,

$T_2 = \{1.0 \times 10^{-4}, 9.9 \times 10^{-4}, 1.0 \times 10^{-9}, 1.0 \times 10^{-11}\}$,
$T_3 = \{1.0 \times 10^{-3}, 9.9 \times 10^{-3}, 1.0 \times 10^{-8}, 1.0 \times 10^{-10}\}$
Same as previous, but with different time windows and gaps in data points.

5. $T_0 = \{1.0 \times 10^{-6}, 9.9 \times 10^{-3}, 1.0 \times 10^{-8}, 1.0 \times 10^{-8}\}$
   The whole interval spanned by 1 STS only, with $N \approx 1,000,000$ time bins. No gaps in data points.

6. $T_0 = \{1.0 \times 10^{-6}, 9.9 \times 10^{-3}, 1.0 \times 10^{-8}, 1.0 \times 10^{-11}\}$
   The whole interval spanned by 1 STS only, with $N \approx 1,000,000$ time bins. Large gaps between data points.

The first test case contains the actual set of STS used in a simulation during the design phase. The other test cases add different challenges on the algorithms, e.g. increased time bins or varying gaps between time bins. In table 5.3 we can see that the approach with the calculative binning works best for most test cases, especially when larger amounts of time moments are used. Only for the first test case (#1), the usage of binary search is superior, where the performance compared to calculative binning is 1.15×. Here, only a considerably low number of time moments has been deployed (36 time moments, 9 for each STS).

Alg/TC	BS	IS	JS	CB	BSwI	JSwI	ISwI
#1	**15.81**	41.05	32.27	18.29	20.79	29.63	29.35
#2	70.13	85.85	310.57	**19.52**	42.28	30.48	27.70
#3	170.75	452.21	3641.59	**45.31**	184.01	382.68	205.03
#4	175.48	468.20	3661.34	**57.65**	191.20	394.67	210.18
#5	155.96	35.27	1313.64	**19.33**	183.31	315.00	73.17
#6	150.94	32.07	1218.38	**17.72**	163.45	276.97	60.69

Table 5.3: Benchmark of multiple algorithms (*Alg*) to determine appropriate bins for time-dependent data structures and various test cases from real simulations. The results of the benchmark are given in milliseconds. The best results are highlighted in bold for each test case (*TC*).

5.3.6 Conclusion

Overall, the calculative binning proved to be the most reliable method. The interpolation search has shown positive characteristics in many, in particular the artificial test cases. If test cases are not set up properly, it can lead to abysmal results, making it largely dependent on the user, where using binary search as a common approach was the most reliable and was therefore used for the first update of the official Molflow release for versions 2.9+ .

We provided a brief outlook, how various search algorithms perform as well as an approach that makes use of the original description used to generate time bins. The initial improvements that could be made by adding a slight constraint to the definition of Specified Time Series in addition with binary search lead to speedups of 9× for a full simulation run. By coupling these efforts with calculative binning, which showed the best performance when individually benchmarking the lookup algorithms, the simulation performance can be further improved. At the time of finalizing this study, the algorithm was not yet part of the released Molflow code. As calculative binning is essentially $O(1)$[1], we wouldn't expect major improvements to follow, without changing the underlying architecture. Another improvement that is left to be investigated for time-dependent simulations is the deployment of adaptive or auto-calculated bins. Right now, the user has to define a set of STS by himself, which requires some trial and error or apriori knowledge to find good values. In many cases, when time events don't happen uniformly and users require higher resolution (more bins) in some time intervals, a bad setting can lead to unnecessary large bin allocations. Further, with the current implementation it is not possible to restrict time events to particular facets. Time-dependent statistics may not be required for all, but only for a small subset of facets, which could further reduce the space complexity and performance of the overall simulations, as bin lookups would not need to happen for all Monte Carlo events.

5.4 Neighbouring Facets

The application uses the neighbourhood information of the facets inside a geometry to apply a variety of functions, e. g. the Smart Selection[2] or for the collapse of facets. Intuitively the relationship of two facets can be determined by finding whether they are sharing a common edge or not. We define a common edge as such that the two edges are defined by the same two points.

5.4.1 Constraints

An important constraint for the definition of a common edge in Molflow is that the orientation is taken into account. This attribute is important, when dealing with the properties of physical simulations. In Molflow, the orientation of a facet is embedded in the order of its vertices. The normal of a facet is given by the

[1]With S the number of STS, which is usually low, it would be $O(\log S)$.
[2]Smart Select allows users to select connected facets by their angular relationship.

left-hand rule. So, in the example in figure 5.5, the same triangle can face either direction. In the case of clockwise order, the triangle faces the reader, and with counterclockwise order it faces in the opposite way. A special case are facets that are two-sided. Such facets are independent from the vertex order and explicitly face both directions. Two facets can only have a common edge, when the orientation of the edge following the same points is the opposite. Consider three facets: facet F_1 with edge $e_1 = (\mathbf{p}_1, \mathbf{p}_2)$, facet F_2 with edge $e_2 = (\mathbf{p}_2, \mathbf{p}_1)$, and facet F_3 with edge $e_3 = (\mathbf{p}_1, \mathbf{p}_2)$. Here, e_1 and e_2 can be considered a common edge, as they follow the same points with differing orientations. Therefore, e_1 and e_3 do not constitute a common edge. In this situation, they have the same orientation, and the facets do not enclose a volume when combined.

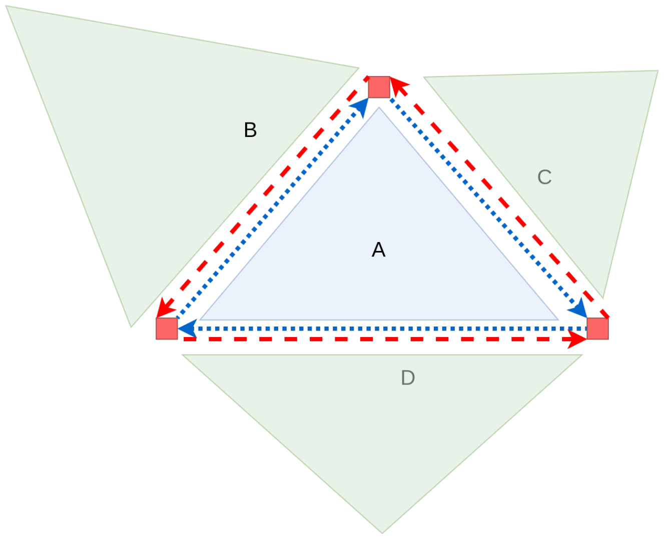


Figure 5.5: Mesh with a reference triangle A and its three neighbours B, C, D. "Physical neighbours" depend on the orientation, which in Molflow is defined by the order of vertices, either clockwise (blue) or counter-clockwise (red).

5.4.2 Algorithm : List-based

The introduction of the Smart Select feature in Molflow 2.7 implemented a simple algorithm to identify and label common edges by comparing each individual facet and their edges. Given a list of facets $\mathcal{F}$, where each facet is described by the corresponding list of vertices. The algorithm links and labels two facets as sharing a common edge when for two consecutive vertices $\mathbf{v}_{i,p}, \mathbf{v}_{i,p+1}$ of facet $F_i \in \mathcal{F}$ and for two consecutive vertices $\mathbf{v}_{j,q}, \mathbf{v}_{j,q+1}$ of facet $F_j \in \mathcal{F}$ the property:

$$\mathbf{v}_{i,p} = \mathbf{v}_{j,q+1} \wedge \mathbf{v}_{i,p+1} = \mathbf{v}_{j,q} \tag{5.4}$$

holds true. This property is critical in determining the orientation of an edge. In our notation, for $\mathbf{v}_{i,p}$ subscripts i are used to denote the index of a facet, while p indicates the position of a vertex within that facet. For a facet F_i with P vertices, the indexing of vertices is cyclic. Therefore, for the last vertex $\mathbf{v}_{i,P-1}$, the next vertex $\mathbf{v}_{i,P}$ is actually the first vertex $\mathbf{v}_{i,0}$. However, for a large number of edges, this proved to be quite inefficient, since the time complexity is $O(E^2)$, where E is the number of all edges (technically all local vector pairs) describing a geometry. This arises from the necessity to compare each edge with every other edge.

In this work, we developed a new algorithm that reduces the complexity of building the data structure, that is keeping the neighbourhood information, to $O(E)$, sorting the input data in $O(E \log E)$, with a lookup performance of $O(1)$ by storing all neighbours directly. Based on the same lookup algorithm, we further developed and investigated another approach that creates a feasible data structure in $O(E)$ using a hashmap.

An optimised algorithm will work with the following steps. First, we create a list containing all edges $e = \{\mathbf{v}_1, \mathbf{v}_2\}$. For that, we go through all facets and create an edge from their corresponding vertices, including the edge connecting the last and the first vertices (in $O(E)$). In addition to saving the corresponding facet ID to the edge, we add an additional property SWAPPED to allow ordering of vertex IDs without losing track of their original order. This is important to maintain the constraint that was explained in chapter 5.4.1. Given two consecutive vertices $\mathbf{v}_{i,p}, \mathbf{v}_{i,p+1}$, the property resolves to

$$\text{SWAPPED} := \begin{cases} 1, & \text{if } \mathrm{ID}(\mathbf{v}_{i,p}) > \mathrm{ID}(\mathbf{v}_{i,p+1}) \\ 0, & \text{otherwise}\,. \end{cases} \tag{5.5}$$

Here, $ID(\circ)$ denotes the ID of a vertex from the global list of vertices. For an edge with the SWAPPED property, the vertices of the edge are swapped: $e = \{\mathbf{v}_{i,p+1}, \mathbf{v}_{i,p}\}$.

Next, having obtained a list of all edges of the geometry, the list is sorted by lexicographical ordering with respect to the corresponding vertex IDs (in $O(E \log E)$). Given two edges e_1, e_2, the edge e_1 will be the first if the following condition is met:

$$\mathrm{ID}(\mathbf{v}_{1,1}) < \mathrm{ID}(\mathbf{v}_{2,1}) \vee \big(\mathrm{ID}(\mathbf{v}_{1,1}) \equiv \mathrm{ID}(\mathbf{v}_{2,1}) \wedge \mathrm{ID}(\mathbf{v}_{1,2}) < \mathrm{ID}(\mathbf{v}_{2,2})\big)\,. \tag{5.6}$$

Having obtained a sorted list, we can simply iterate through the list and compare one entry with the neighboring entries. We do this by using a pursuing and a leading pointer. The former is the starting point of a search. All following entries, using the leading pointer ce, that have the same vertex are compared to identify a potential common edge. The pursuing pointer ce_2 will describe the first edge, the leading pointer the second edge. Keeping in mind the SWAPPED property and the correct order of the vertices (5.4) a common edge can be marked e.g. by adding the corresponding facet ID from the second edge to the first. When the pursuing pointer encounters an edge with only one facet entry, this entry can be deleted. If the pursuing pointer ce_2 identifies a common edge and the original edge on the main pointer ce is linked to more than one facet, a duplicate of this edge is created. This duplicate is then modified to include only the first facet. The facets from the subsequent edge (ce_2) are merged into this duplicate. This newly formed edge is added to a separate list, CCE, which is used for managing situations where an edge is common to more than two facets. Once duplicates are eliminated from CCE, this list is then combined with the existing list CE. The routine COMBINE_EDGES to find and return a list of common edges, based on a sorted input list is, is depicted in algorithm 4. This routine is $O(E)$ for the amount of edges E in the data set. Leaving the whole algorithm at $O(E \log E)$ complexity, which is primarily due to the sorting operation of the edges as an inherent requirement[3].

A similar solution was proposed by Qi et al. (2020), where the neighbouring elements were defined having a shared vertex v . An adjacent element could then be found by first sorting a list of tuples $\{v, F\}$ for the whole mesh, where a tuple describes a vertex v that is part of facet F. This is followed by a segmented scan. Due to the constraint imposed for the strict definition of a common edge, this approach cannot be applied without any modifications.

For further post-processing, the angle between each pair of facets sharing a common edge can be calculated using the corresponding facet normals:

$$cos\theta = \frac{|N_1 \cdot N_2|}{|N_1||N_2|}\,. \tag{5.7}$$

This angle can then be used for features like Smart Selection, where adjacent facets are selected on the basis of their angle.

5.4.3 Algorithm : Map-based

As the algorithm primarily relies on the performance of insert and delete operations to directly modify the given input, we evaluated its performance on three different C++ standard containers. A basic implementation based on vectors is limited by the efficiency of deletions ($O(N)$, for N data points), which is carried out frequently in algorithm 4. Using a list for efficient delete operations ($O(1)$) is evidently beneficial. We propose another solution based on a map or a hashmap. Here, an edge $e =$

[3]Note: As part of this work we did not experiment with other sorting algorithms such as radix sort, which could potentially reduce the time complexity further to $O(E)$.

Algorithm 4 Common Edge finding algorithm COMBINE_EDGES(CE).

```
 1: function COMBINE_EDGES($CE$)
    ▷ Return list of common edges $CE$ from a list of all edges in a geometry $\Omega$
    ▷ Loop over all edges via edge iterator $ce$
    ▷ if multiple shared facets are found,
    ▷ create an external extra facet for later merging into main list
 2:     $n_{next} \leftarrow 1$                          ▷ Count how far ahead trailing iterator is
 3:     while Edge iterator $ce$ has not reached the end do
 4:         $ce_2 \leftarrow next(ce, n_{next})$        ▷ Set trailing iterator to be $n_{next}$ steps ahead
 5:         if $ce_2$ is last edge in list then
 6:             if $ce$ has no linked facet then
 7:                 Delete $ce$ from list
 8:             end if
 9:             $ce \leftarrow$ next($ce$)
10:             $n_{next} \leftarrow 1$
11:         else if Vertices of $ce$ and $ce_2$ are identical then
12:             if One of $ce$'s facets is identical to $ce_2$'s facets AND $ce$'s and $ce_2$'s
    SWAPPED is different then
                    ▷ Common edge found
13:                 if $ce$ has only 1 facet then
14:                     Add $ce_2$'s facets to $ce$'s linked facets
15:                 else
16:                     Create copy of $ce$ and only keep the first facet
17:                     Add $ce_2$'s facets to $ce$'s linked facets
18:                     Keep copy in extra list $CCE$
19:                 end if
20:                 $n_{next} \leftarrow n_{next} + 1$
21:             end if
22:         else
23:             Delete $ce$ from list
24:             $n_{next} \leftarrow 1$
25:         end if
26:     end while
27:     Remove duplicates in $CCE$
28:     Append $CCE$ to $CE$
29: end function
```

$\{\mathbf{p}_1, \mathbf{p}_2\}$ is inserted with the key k and the hash function

$$k = (ID(\mathbf{p}_1)\texttt{<<}16) \quad | \quad ID(\mathbf{p}_2)\,. \tag{5.8}$$

Assuming a 32 bit hashkey k and a IDs for each point using at most 16 bit, first, we do a leftwise bit shift $<<$ by 16 bits for the ID of the point p_1. Now, the ID is represented by the 16 most significant bits of the 32 bit hashkey, the 16 least significant bits are all zero. A bitwise OR operation | is then used to ensure that the ID of p_2 occupies the 16 least significant bits of the hashkey. This unique hashkey k thus encodes the IDs of both points in the edge. In this case, it is not necessary to swap the vertices. For each entry in the map, we save all facet IDs that contain the corresponding edge. Later, to find the common edge, the hash can simply be reverted. First, we can reverse k:

$$ID(\mathbf{p}_1) = (k\texttt{>>}16)\,, \tag{5.9}$$
$$ID(\mathbf{p}_2) = (k\%2^{16})\,. \tag{5.10}$$

Next, we build a reverse key k_r

$$k_r = (ID(\mathbf{p}_2)\texttt{<<}16) \quad | \quad ID(\mathbf{p}_1) \tag{5.11}$$

and use it directly to lookup any potential facet sharing a common edge. The proposed method works only when at most 2^{16} vertices are used to describe the geometry. The corresponding hash functions can be easily modified to account for 2^{32} vertices, when a 64-bit hashkey is used. For our tests, the proposed hash functions are sufficient.

5.4.4 Benchmark

The correctness of our solution in terms of the added constraints has been evaluated first with a test suite inside Molflow that is comparing the results of all proposed algorithms against the original algorithm used in Molflow 2.7 . The original benchmark, while a naive implementation, respected the required constraints for the definition of a common edge.

In total, the following algorithms have been implemented:

- Common Edges - original (Molflow 2.7)
- Common Edges - Vector-based
- Common Edges - List-based
- Common Edges - Hashtable-based

- Common Edges - Map-based

In addition, this test suite is used to benchmark all algorithms. Here, we consider 3 geometries of different levels of complexity in terms of the amount of vertices and facets and their interconnectivity. The geometries are elaborated in chapter 4. One is a part from CERN's FCC concept (see figure 4.2) with 292478 vertices, the other is part of the Hollow Electron Lens used for the LHC upgrade with 39575 vertices (see figure 4.4) and the other is an RF cavity with 55018 vertices (see figure 4.5).

From the results provided in table 5.4, we can see that the implementation using a doubly-linked list (default C++ implementation) yields the best results. Due to the better access time, the implementation using a map container (access complexity $O(1)$) was expected to be an improvement to the other implementations, which was not the case using the three provided real life examples. The original algorithm was expected to perform the worst due to its $O(F^2)$ complexity. The vector implementation was only marginal enhancement, due to expensive operations to modify the structure. Due to the performant results, in Molflow 2.9, the algorithm based on lists as an efficient data structure has been used.

Alg/TC	**#1**	**#2**	**#3**
Original	135.2876	1030.2534	4557.4306
Vec	21.2695	86.4227	1131.1273
List	0.1003	0.2173	0.6991
Hash table	0.1620	0.3368	0.8497
Map	0.1129	0.2516	0.5357

Table 5.4: Benchmarks provided for the different algorithms and three different geometries. The values show the duration in seconds for the entire algorithm: construction and common edge lookup.

5.4.5 Conclusion

In this chapter, we discussed algorithms for identifying neighbouring facets in Molflow, emphasizing the significance of facet orientation and shared edges in physical simulations as a particular constraint. Through a detailed exploration of various algorithms designed for efficient recognition of common edges, we have demonstrated how the additional constraints play a role in defining their relationship.

Our comprehensive analysis concluded with the development of an optimized algorithm, which significantly enhances the performance of this algorithm, which serves several roles in Molflow. Tools such as the Smart Selection feature in Molflow, which leverages the angular relationship between connected facets, are practical examples that directly benefit from the enhanced performance. Furthermore, using the proposed technique, we also improved other algorithms. Conversion from triangle meshes to polygon meshes incorporates the routine to collapse facets that are coplanar to each other for a certain threshold, which utilises the proposed routine for faster execution. Determining neighbourhood relationships efficiently was a requirement for a newly developed techniques for our GPU kernel, elaborated in chapter 5.4, that utilizes these information.

6 Iterative Simulations

Iterative simulations in Molflow enable the simulation of vacuum chamber aspects not previously possible with existing tools. In this context, a simulation $\mathbf{S}$ is transformed into a pipeline of sub-simulations $\mathbf{s_i}$ (with $i \in [0, .., n-1]$). The output of simulation $\mathbf{s_i}$ serves as input for simulation $\mathbf{s_{i+1}}$, with $\mathbf{s_0}$ working on the initial input and $\mathbf{s_{n-1}}$ providing the final output.

This chapter outlines the motivation for and theoretical foundation of iterative simulations, developed as part of this project. A complete implementation of the iterative algorithm was not possible for this thesis due to time constraints and other priorities. However, individual components serving multiple purposes, which are essential for efficient and accurate iterative simulations and also serve other purposes, were implemented independently. These components will be discussed in this chapter, particularly focusing on specialized convergence criteria.

It should be noted that in parallel to this thesis, different developers at CERN initiated the development of iterative simulations using the features of Molflow's command line interface, which was developed as part of this work. This code was released as a public software called VacuumCOST as a set of Python scripts by CERN (Henriksen, M. Ady, and R. Kersevan, 2023). This code was validated against a set of existing benchmarks[1].

6.1 Motivation

Iterative simulations hold particular interest in the context of particle accelerators, as vacuum properties are not always static or time-dependent. For some types of pumps, properties change over their lifespan due to various factors. This can occur, for example, due to constant bombardment of inner surfaces by electrons, photons, and ions, leading to desorption of gas molecules. Both the pumping speed and the sticking factor of NEG (Non-evaporable getter) coatings decrease as the surface coverage of a gas increases, which Chiggiato and Costa Pinto (2006) elaborated. As an example, figure 6.1 shows how the pumping speed of different gases changes with the surface coverage of carbon monoxide (CO). This effect cannot be modelled directly with the simulation tools within Molflow. Time-dependent simulations (which have

[1]Source: The whitepaper on Molflow+ iterative simulations was not publicly available at the time of finalizing this thesis.

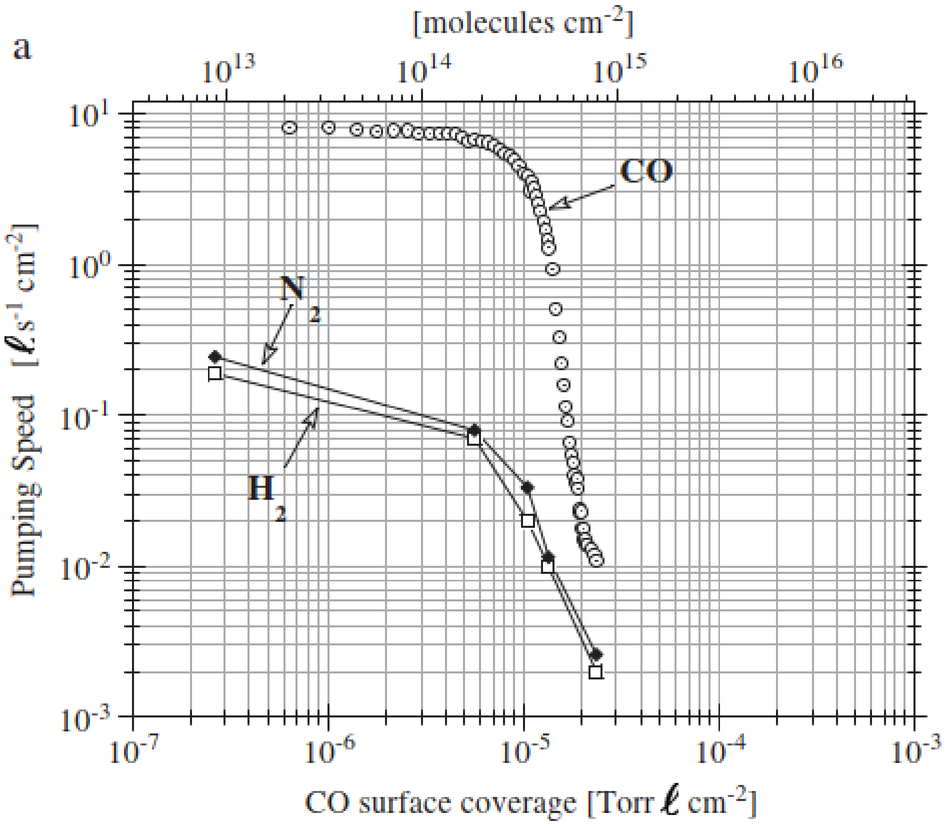


Figure 6.1: Pumping speed dependent on CO surface coverage (Chiggiato and Costa Pinto, 2006).

been discussed in more detail in chapter 5.3) allow to model certain parameters as a function of time. Here, we require that a parameter (pumping speed) changes in the system as an effect of the physical events. A new type of simulation would require results from previous simulations, in particular the amount of particles that have already been pumped, to estimate the surface coverage on a facet.

A simplified algorithm for an iterative simulation run would work like this:

1. Launch simulation for a time step t_n.
2. Stop with arbitrary stopping criterion.
3. Use results from time step t_n for t_{n+1}.

The initial prototype should explore the idea by launching simulations with time steps t_n of a fixed, user-defined time step size $\mathrm{d}t$. The potential to automatic calculation of the time step size warrants further investigation, given that there is no known method of mathematically predicting the rate change between surface coverage and pumping speed. Stopping criteria can be initially set manually as well, e.g. with a pre-defined amount of desorbed particles, more advanced stopping or convergence criteria are discussed later in chapter 6.2.

To track the surface coverage a fine-grained counting structure is necessary to reduce the numerical error. This is because facet saturation is unlikely to happen homogeneously. Intuitively a *texture* mesh can be deployed for a single facet to keep track of the incident and adsorbed particles. In Molflow this can be realized with a new type of texture, following the principles of existing textures used as a hit counter, which were discussed in chapter 2.2.3. Such a texture needs to exist twice, one to keep track of the change of saturation and another that can be used to retrieve local sticking factors. In figure 6.2 an example for a Molflow geometry with a textured facet is shown.

A good balance will have to be achieved, without putting an impact on the simulation requirement for memory. Such textures should initially be simply manually defined, where auto-tuning could be resolved e.g. by adjusting the mesh size depending on the preceding simulation.

To recap the potential challenges for an iterative simulation, first, the time step size $\mathrm{d}t_j$ of each iteration step j has to be determined. Preferably, a dynamic step size is used, as the pumping speed does not decrease linearly. Only for initial testing, a fixed time step size $\mathrm{d}t_j = \mathrm{d}t \quad \forall j$ for the whole simulation will be sufficient. Next, as an iterative simulation is essentially a chain of consecutive time-dependent simulations, the individual simulation steps have to be stopped at a reasonable time. Intuitively, a convergence criterion that determines sufficiently converged facet saturation levels to stop the simulation should be deployed. Fixed desorption limits will likely overshoot or undershoot, leading to either underdeveloped solutions or

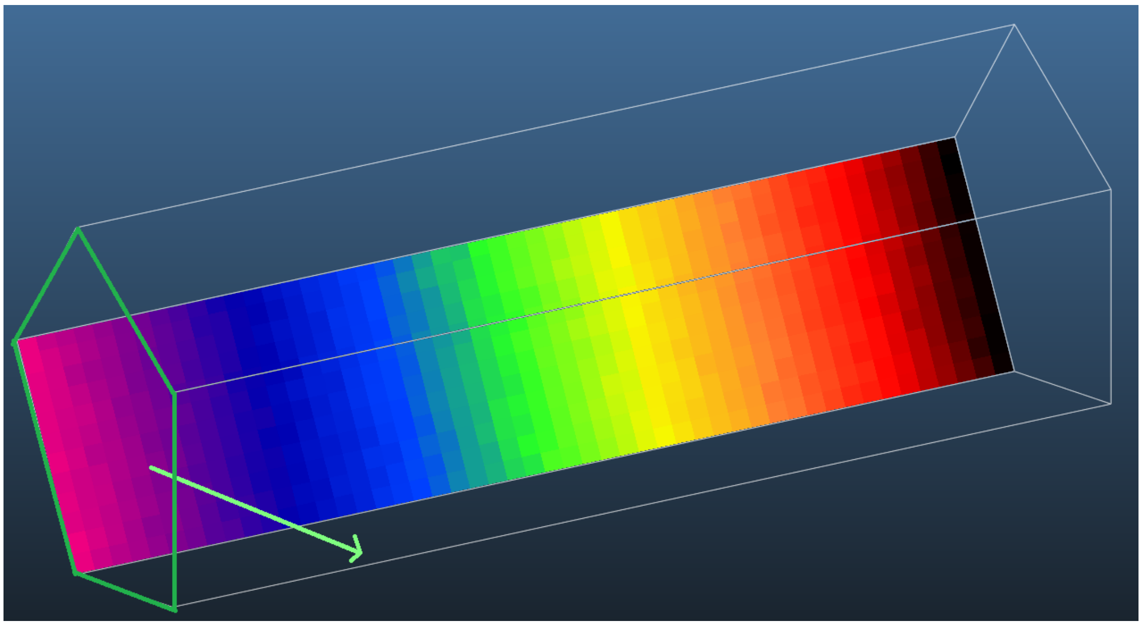


Figure 6.2: A Molflow geometry with a textured surface. Here, it is representing the amount of Monte Carlo events, decreasing from left (purple) to the right (black). Gas inflow is coming from the left. The desorbing facet is highlighted in green. An exemplary particle is sketched as a green arrow.

wasteful use of computer resources. Lastly, mesh resolutions to track saturation levels have to be carefully, ideally auto-, adjusted. To identify an ideal method to design the mesh resolution with minimal impact on memory demand yet with yielding good performance, a more in-depth study has to be conducted once the functional aspects of the iterative simulations are fully implemented.

6.1.1 Iterative algorithm

This chapter will outline the general concept about the iterative algorithm that was initially sketched as part of thesis.

The data structures for the simulation side have been worked out as follows. A facet needs to contain a new type of persistent texture, where the individual saturation levels of each element are converted into sticking coefficients, leading to a Sticking Map. With a Sticking Map, the changed sticking coefficients could be fetched with higher precision than for a single value per facet. In parallel, a regular counting texture needs to keep track of particle hits, from which the surface saturation (that is pumped molecules/cm^2) can be derived. This counting texture can then be used as input for the Sticking Map of the next iteration.

To setup an iterative simulation, a parameter distribution such as these found by Chiggiato and Costa Pinto (2006) have to be imported and set as a material property $s(\sigma)$ for each concerned facet. An iterative simulation would then start for a time interval $[t_0 = 0, \mathrm{d}t_0)$. The first iteration step would begin by automatically identifying a suitable $\mathrm{d}t_0$ or manually setting the value by the user e.g. by a fixed time $\mathrm{d}t_i = \mathrm{d}t_{max} \quad \forall i$. This would then launch an iteration step with a time-dependent simulation from $[t_0 = 0, \mathrm{d}t_0)$. On collision, for facets with the new texture type, the sticking factor is chosen according to the distribution $s(\sigma)$ given the value σ for the surface saturation (pumped molecules / cm^2) from the corresponding element from the sticking map, to either pump or reflect the particle. In regular intervals, the simulation will check against an end criterion and eventually stop the current iteration. Trigger for this criterion could be a fixed amount of desorptions, a fixed simulation time or the fulfilment of a convergence criterion (see chapter 6.2).

For the next i-th iteration step, the procedure is essentially the same. The sticking map will be updated according to the results of the $(i-1)$-th step. Next $\mathrm{d}t_i$ will need to be automatically identified or again taken from manual user input. The next iteration step then starts a time-dependent simulation with particle time from $[t_i, t_i + \mathrm{d}t_0)$. Note, that depending on the design, the behaviour of particles from the previous simulation that went beyond t_i should still be considered.

Following either a static or a dynamically found time-window, an iteration step would correspond to a time-dependent simulation (see chapter 5.3) with a single time moment for the time intervals mentioned. Only particles contributing to this time moment should actually have an effect on the simulation results. An empirical trial has to find out, whether the simulation results will be strongly affected by neglecting the cut-off particles – those who exit the active time-window –, or whether these have to contribute to the counters of the $(i + 1)$-th simulation step.

6.2 Convergence

With the convergence plotter[2], a new set of statistical data is available to the user to evaluate the results of a simulation. In figure 6.3 we show the pressure profile for all side facets of a cylindrical vacuum tube (as mentioned in chapter 4.3), where one can easily see that the value converges with an increasing amount of desorptions N. Data on convergence is typically used to estimate the precision of simulation results. It is up to the user to define a suitable term that may significantly represent whether a system converges or not. Commonly, Molflow users are interested in the pressure values or transmission probability. For a particular facet i, the transmission probability is defined as the ratio of the number of particles absorbed by this facet i to the total number of particles desorbed from all facets.

Expressing convergence is in particular of interest for problems, where multiple dependent simulations are chained together and where a good understanding of the convergence allows to characterise when the transition between the individual simulation stages can be applied without carrying significant bias over. This is not only the case for the before-mentioned iterative simulations, where the input of the i-th simulation is used for the $(i + 1)$-th simulation, but also for simulations for photon-stimulated desorptions utilising cross-software simulations, where a Synrad simulation result is used as the input for a Molflow simulation (see chapter 2.2.9).

First, we give a brief overview of the statistical properties of Monte Carlo, based on the deep introduction to the topic of statistics by DeGroot and Schervish (2012) and Monte Carlo methods by Kroese, Taimre, and Botev (2011). This is followed by a discussion of stopping criteria, which are useful markers upon which a simulation can be stopped.

6.2.1 Basics

The main goal of a Monte Carlo simulation is to approximately compute a mean value μ with the variance σ^2, in Molflow depicting a physical parameter value (see chapter 2.2.3 for a detailed discussion about hit counters), for which a direct calculation or analytic solution is not computationally feasible. As we can only get approximative results based on a set of sample data, we try to determine the estimator for the mean $\bar{x}_n$ and the estimator for the variance s_n^2 for n samples.

[2]Released 11/2020 with Molflow 2.8.4 .

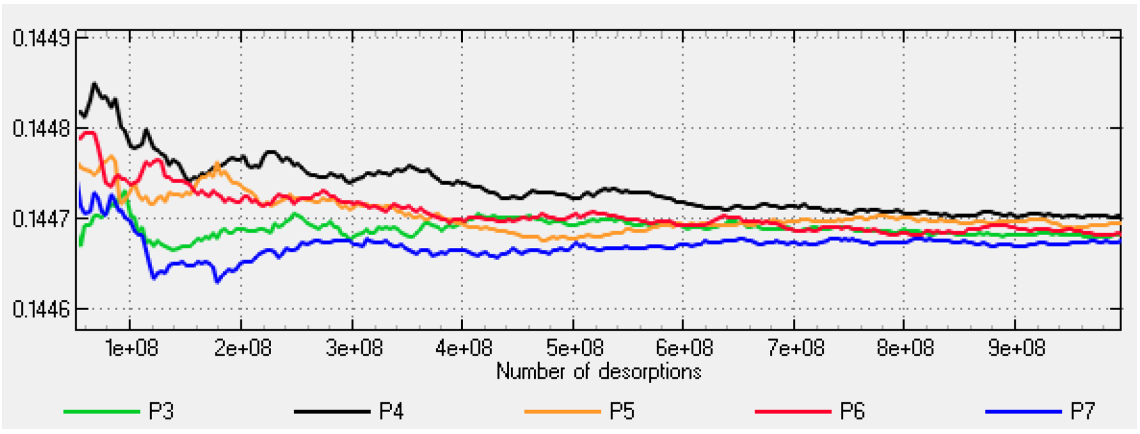


Figure 6.3: Convergence profile for pressure values of multiple facets for a vacuum pipe geometry. Each profile shows the computed pressure value (y-axis) after N desorptions (x-axis). The profiles are converging towards the same value with increasing desorptions. This is expected as the pipe geometry has radial symmetry and its inflow parameters are constant across all positions.

Given a random experiment of n i.i.d. (independent and identically distributed) samples[3] $x_1, \dots, x_n$, the sample mean can be computed as

$$\bar{x}_n = \frac{1}{n}\sum_{i=1}^{n} x_i \,. \tag{6.1}$$

Here, each sample x_i refers to an event of a particle history. For the i.i.d. variable x_i we have

$$var\left(\sum_{i=1}^{n} x_i\right) = n\sigma^2 \,, \tag{6.2}$$

with the standard deviation σ, and therefore a variance for the sample mean as

$$var(\bar{x}_n) = var\left(\frac{1}{n}\sum_{i=1}^{n} x_i\right) = \left(\frac{1}{n}\right)^2 var\left(\sum_{i=1}^{n} x_i\right) = \frac{1}{n^2}(n\sigma^2) = \frac{\sigma^2}{n} \,. \tag{6.3}$$

Therefore, the standard deviation from the sample mean to the real mean should be proportional to $\sigma/\sqrt{n}$, from which we can derive a convergence rate of $1/\sqrt{n}$ for asymptotically increasing n.

[3]It is impossible to treat individual Monte Carlo events as i.i.d.. Each tracked particle leads to a series of dependent hits, so only the full histories of each particle can be i.i.d..

The variance σ^2, which represents the spread of the samples, can then be estimated as

$$\sigma^2 \approx s^2 = \frac{1}{n-1} \sum_{i=1}^{n} (x_i - \bar{x}_n)^2 \,. \tag{6.4}$$

The efficacy and precision of a Monte Carlo method in estimating a parameter can be related to the rate of convergence. It can be described by the Central Limit Theorem (CLT). It describes how the distribution of sample means approaches a normal distribution for a rising amount of samples n. Given a model with a non-fluctuating mean value μ for some characteristic parameter and the corresponding standard variation σ, for a sample size $n \to \infty$ a Monte Carlo estimator can be given as $\bar{x}_n \to \mu$ with the corresponding standard deviation $s = \frac{\sigma}{\sqrt{n}}$.

In the context of Monte Carlo estimation, the sample mean $\bar{x}_n$ can be calculated using equation (6.1), where x_i represents independent and identically distributed samples obtained through Monte Carlo simulations. The Central Limit Theorem asserts that as the sample size n increases, the distribution of the sample means approaches a normal distribution. This leads to the standardization of the sample mean with the formula:

$$\mathsf{z}_n = \frac{\bar{x}_n - \mu}{\sigma \sqrt{n}} \,, \tag{6.5}$$

where z_n is z-score for the n-th sample. The z-score indicates how many standard deviations a data point differs from the mean of a distribution.

Understanding the accuracy and reliability of the results of Monte Carlo simulations is crucial. By understanding how the sample means distribute and converge, an optimal point can be determined to stop a simulation. Next, we will elaborate on stopping criteria in general, before more suitable convergence criteria are studied and evaluated to assist the users e. g. with an automatic stopping criterion.

6.2.2 Stopping criteria

Stopping criteria don't automatically translate to convergence criteria, as they are usually more simple in nature and they do not try to estimate convergence in a particular sense. Stopping criteria typically fall into two categories: fixed criteria and dynamic criteria. Fixed stopping criteria are often based on predetermined knowledge, e.g. for the total sample size or the total simulation runtime. In contrast to that, there are dynamic criteria, e.g. those that try to determine the sample size by using knowledge gained from the simulation itself. This is particularly relevant in iterative simulations, where the simulation results of a preceding simulation are used as input for the next iteration. In such cases, ensuring that each simulation stage has reached a reasonable level of convergence is crucial. This is a goal that fixed criteria can only achieve by greatly overestimating the required simulation runtime or the necessary sample size. Therefore, the selection of appropriate stopping criteria must balance the computational efficiency with the assurance of statistical accuracy.

6.2.3 Convergence criteria

It is very difficult to make absolute statements when a random experiment has sufficiently converged. Intuitively, one would apply some of the more common statistical tools. The Coefficient Of Variation (COV)

$$\varepsilon_{COV} = \frac{s}{\bar{x}}\,, \tag{6.6}$$

is a measure to determine the extent of data fluctuation from the mean of all samples. Here, s is the estimated standard deviation (derived from equation (6.4)) and $\bar{x}$ is the mean (from equation (6.1)). Compared to the direct usage of the standard deviation, the COV is a unitless scalar showing the standard deviation in relation to the mean. A high COV means that the data has a high degree of dispersion around the mean, whereas a low COV means that the sample points are closer.

The Mean Squared Error (MSE) can be used to quantify the accuracy of an estimator. MSE calculates the average of the squares of the errors, that is, the average squared difference between the estimated values and the actual value. It can be described by:

$$\mathrm{MSE} = \frac{1}{n}\sum_{i=1}^{n}(\bar{x}_i - \mu)^2\,, \tag{6.7}$$

where $\bar{x}_i$ represents the estimated values from each sample, and μ is the true value of a model. It is particularly useful in assessing the performance of an estimator: a lower MSE indicates a higher accuracy, suggesting that estimator closely predicts the true values. It is a useful concept that can be adapted for Monte Carlo simulations, where the true value μ is usually unknown. Here, MSE is not used to compare estimators against actual values, but rather to measure the stability and convergence of the simulations over successive iterations. We can modify MSE (6.7) to compare the results of consecutive iterations of a simulation:

$$\varepsilon_{MSE} = \frac{1}{n}\sum_{i=1}^{n}(\phi_i^k - \phi_i^{k-1})^2\,. \tag{6.8}$$

Here, ϕ_i^n and ϕ_i^{n-1} represent the values or estimators at the k-th and $(k-1)$-th iterations, respectively. The modified MSE enables us to estimate the degree of change between iterations. As the simulation progresses, a decreasing MSE suggests that the results are converging, indicating that the simulation is stabilizing and approaching a consistent outcome. While the MSE can give a good estimate for the convergence of an MC simulation, there are other statistical tools to quantify the reliability and accuracy of a result.

6.2.4 Confidence intervals

Confidence Intervals (CIs) provide a clearer understanding of the quality of the simulation results. With a specified degree of confidence, they allow us to quantify the extent to which the simulation results are converged. The confidence interval for n samples is given by:

$$\mathsf{I} = (l_\alpha, u_\alpha) = (\bar{x}_n - \mathsf{z}_{\alpha/2}\frac{s}{\sqrt{n}} \quad , \quad \bar{x}_n + \mathsf{z}_{\alpha/2}\frac{s}{\sqrt{n}})\,, \tag{6.9}$$

with the sample mean $\bar{x}_n$, the sample standard deviation s_n and z_k representing the z-score that corresponds to the k-th percentile of the standard normal distribution. The z-score indicates how many standard deviations a data point is from the mean of the distribution. α represents the probability that the true value is outside the confidence interval.

To retrieve the standard deviation for a traditional confidence interval, the estimator of the sample variance s_n^2 given by equation (6.4) is required. Calculating the estimator s_n^2 from scratch for every new sample can be computationally expensive, making it impractical for Molflow's Monte Carlo simulations. This is because by default Molflow does not generate a stream of individual histories x_i, but instead sequentially updates the data, keeping only the previous sample mean $\bar{x}_{n-1}$ and the latest history x_n. Because of this, we can not compute the variance (6.4) directly.

An efficient alternative is to use an online algorithm to iteratively calculate $\bar{\mu}_n$ and s_n^2, thus reducing computational overhead. The algorithm updates these values with each new data point:

$$\bar{\mu}_{n+1} = \bar{\mu}_n + \frac{X_{n+1} - \bar{\mu}_n}{n+1}\,, \tag{6.10}$$

$$s_{n+1}^2 = \left(1 - \frac{1}{n}\right) S_n^2 + (n+1)\left(\bar{\mu}_{n+1} - \bar{\mu}_n\right)^2\,. \tag{6.11}$$

This algorithm by Welford (1962) allows for assessing the convergence and precision of the simulation without the significant memory and computational costs associated with the traditional method to iteratively update the variance (6.4).

A convergence criterion based on CIs can be defined by considering the width W_n of the confidence interval for n samples

$$W_n = 2z \cdot \frac{s_n}{\sqrt{n}}\,, \tag{6.12}$$

and assuming convergence if:

$$W_n \leq \varepsilon\,. \tag{6.13}$$

Here, ε is an absolute error threshold, which has to be carefully chosen. It is more convenient to define the equation based on a relative error threshold ε_{rel}:

$$W_n \leq \varepsilon_{rel} \cdot \bar{x}_n\,, \tag{6.14}$$

where the error is related to the sample mean.

In Monte Carlo simulations, CIs can lead to overly conservative estimates. In particular for high confidence levels (e.g. 99%), this can lead to larger than necessary sample sizes. Other problems, such as the numerical cancellation are discussed by Chan, Golub, and LeVeque (1983). This lead to the investigation of utilizing confidence bands, which are elaborated in the following chapter. Convergence bands remove the necessity of global knowledge and only look at local values throughout a specified band width.

6.2.5 Acceptable Shifting Convergence Band Rule criterion

The Acceptable Shifting Convergence Band Rule (ASCBR) criterion as proposed by Ata (2007) is an efficient convergence criterion for Monte Carlo simulations. Following his analysis, it combines the robustness of traditional confidence interval approaches with a dynamic sampling technique. The approach follows the idea, that measurements have converged, when the most recent samples are close to one of the previous measurements.

A sequence of N sample points is considered *acceptable* or *converged*, if the last ζ samples are within a sufficiently small window around the $(N-\zeta)$-th sample. From a starting point with the sample mean $\bar{x}_j$, for the j-th sample, an interval can be built given the upper and lower limits:

$$U(\bar{x}_j) = \begin{cases} U(\bar{x}_{j-1}), & \delta_j = 0\,, \\ \bar{x}_j + \varepsilon, & \delta_j = 1\,, \end{cases} \tag{6.15}$$

$$L(\bar{x}_j) = \begin{cases} L(\bar{x}_{j-1}), & \delta_j = 0\,, \\ \bar{x}_j - \varepsilon, & \delta_j = 1\,, \end{cases} \tag{6.16}$$

where

$$\delta_j = \begin{cases} 0, & L(\bar{x}_{j-1}) < \bar{x}_j < U(\bar{x}_{j-1})\,, \\ 1, & \{\bar{x}_j \leq L(\bar{x}_{j-1})\} \vee \{\bar{x}_j \geq U(\bar{x}_{j-1})\}\,. \end{cases} \tag{6.17}$$

Here, δ_j is a decision variable that determines whether the current sample mean $\bar{x}_j$ is within the previously set limits. If it is within these limits ($\delta_j = 0$), we keep the limits unchanged. If it is outside ($\delta_j = 1$), we adjust the limits following equations (6.15) and (6.16) by applying a small margin ε to the new sample mean. When evaluating a random variable Z_j with the observed values $Z_0 = 0$ and

$$Z_j = \begin{cases} Z_{j-1} + 1, & \delta_j = 0\,, \\ 0, & \delta_j = 1\,, \end{cases} \tag{6.18}$$

a stopping rule can be defined with:

$$N(\varepsilon, \zeta) = \min\{j : Z_j = \zeta\}\,, \quad j \in \mathbb{N}, \tag{6.19}$$

which represents the total amount of samples acquired before the rule is satisfied. Z_j counts how many consecutive samples fall within the set limits. It starts at 0 and increases by 1 for each sample within the limits. If a sample falls outside, Z_j resets to 0. The stopping rule (6.19) implies, that the simulation stops after finding the j-the sample so that Z_j reaches ζ. The parameter ζ represents the number of consecutive sample points required to fall within the convergence band defined by the parameter ε and equations (6.15) and (6.16). Once this condition is met, we conclude that the sample has converged.

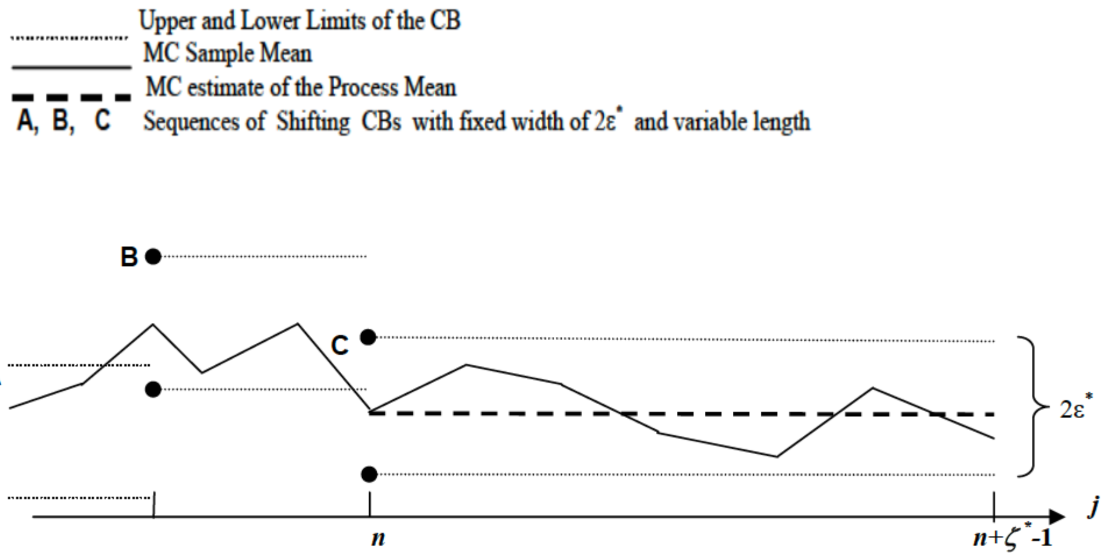

Upper and Lower Limits of the CB
MC Sample Mean
MC estimate of the Process Mean
A, B, C Sequences of Shifting CBs with fixed width of 2ε* and variable length
A
B
C
2ε*
j
n
n+ζ*-1

Figure 6.4: Visualisation of the ASCBR criterion. Here, we see three convergence bands A, B, C, where the dots represent the newly defined lower and upper limits. Here, only in the final convergence band C the samples stay within an error window of size 2ε. The sample mean does not stay within the first two sequences A and B (Ata, 2007).

In figure 6.4 the ASCBR is visualised with three convergence bands A,B and C. The foregoing subsequences A and B did not result in a convergence band of sufficient length to claim convergence confidentially. Here, (6.19) has not been fulfilled due to $Z_j < \zeta\,, \forall j \leq n$. Subsequence C, on the other hand, results in a convergence band that fulfils the stopping criterion (6.19), with enough sample points staying within the convergence band bounds $Z_{n+\zeta-1} = \zeta$.

With a well-chosen convergence band length ζ and half-width of the window ε^* the resulting stopping point $N(\zeta, \varepsilon^*)$ will be very close to that of a N_{CI} stopping criterion following a set confidence interval. The authors describe an empirical approach to determine ζ for a given ε to correspond to a confidence interval. According to an empirical evaluation, the density function $PR(\zeta)$ represents the probability of achieving a convergence band length ζ. The authors approximated this distribution using the logarithmic series variate. For the density function:

$$PR(\zeta) = c^{\zeta} - \ln(1-c) \cdot \zeta^{-1}, \quad \zeta = 1, 2, \ldots; \quad 0 < c < 1 \tag{6.20}$$

they estimated the shape parameter c with:

$$\hat{c} = 1 - \frac{f_1(\zeta = 1)}{\sum_{i=1}^{51} i \cdot f_i(\zeta = i)} \tag{6.21}$$

from their experiments by comparing the probability of achieving a convergence band length of $i = 1$ with the average probability of achieving convergence band lengths $i = 1, 2, \ldots, 51$, where $f_i(\zeta = i)$ denotes the corresponding probability for each convergence band length. They deduced the value of c to be $c = 0.9$ for their selected stochastic processes for three levels of precision by assuming $\zeta > 50$ to be a good value as a rule of thumb. In our own experiments, we couldn't deduce a general shape parameter. We created an implementation of the ASCBR criterion in Molflow and validated how they compare with classic confidence intervals to find whether the criterion is giving comparable results without the extra overhead as described. We provide a rule of thumb on the parameters for achieving convergence in Molflow with the ASCBR confidently. The results are presented in chapter 6.2.7.

6.2.6 Entropy

An alternative to considering local values for the sample mean, such as pressure or transmission probability values, is the usage of Shannon entropy. The idea is to observe how the entropy values change as the simulation progresses. Entropy, in this context, quantifies the uncertainty or randomness within the simulation. It can serve as a statistical measure to evaluate whether a set of sampled values converge towards a stable mean.

In some Monte Carlo simulations, particularly those involving particle transport, such as OpenMC or MCNP, Shannon entropy is employed as a metric (see Brown (2006), Haghighat (2020), and Kumar, Forget, and Smith (2020)) to determine whether a solution is sufficiently converged or not. In these applications, a spatial mesh is employed to bin information about so-called neutron source sites, that are those locations where nuclear fission events occur, leading to the emission of additional neutrons. The entropy is then defined by

$$S(n) = -\sum_{m} p_m(n) \log_2(p_m(n)), \tag{6.22}$$

where n is the number for the current generation, where each generation relates to a fixed amount of events, m is the index for the mesh bin, and $p_m(n)$ is the ratio of neutron source sites in bin m to the total number of source sites. Using Shannon entropy, a stopping criterion is reached when the entropy value stabilizes over successive iterations.

In attempting to adapt this approach for Molflow, the direct information about the relative amounts of events per facet has been used. To capture when then entropy value stabilizes sufficiently, we deploy the ASCBR stopping criterion. This approach did not prove to be feasible in real simulations though, as it lead to irregular spikes in the curve. This irregularity makes it difficult to discern a clear trend towards convergence. It was observed that the individual terms in the entropy calculation often balanced each other, even in simulations with more complex geometries. This issue, also stated by Haghighat (2020), suggests a potential limitation of the entropy-based approach in capturing the nuances of particle transport simulations.

While entropy shows promise as a convergence criterion in certain Monte Carlo simulations, its application in Molflow simulations presents unique challenges. Fluctuations or irregularities in the entropy value can make it difficult to ascertain whether true convergence has been achieved. These challenges highlight the need for further research, such as more refined methods for quantifying events, e.g. using different measures such as pressure values or transmission probabilities for a set of facets.

6.2.7 Empirical validation

To validate both the correctness and efficiency of the ASCBR, we launch several simulations on a set of input geometries for which an analytic analysis or an experimental validation exists. We utilise the cylindrical tube with L/R=10, whose solution for the transmission probability was obtained by Gómez-Goñi and Lobo (2003).

We evaluate how reliable the ASCBR works as a stopping criterion with convergence windows for the relative errors $\varepsilon_{rel,1} = 10^{-4}$ and $\varepsilon_{rel,2} = 10^{-5}$. In table 6.1 we list how many desorbed particles had to be processed to stop and the difference between the analytical solution and the simulation result for 3 runs per test case. For a relative error of 10^{-4}, the results were in the range of ± 0.000031. For 10^{-5}, the results had a tolerance of ± 0.000008. Overall, the ASCBR with the utilized parameters proved to be reliable. It would serve as a good tool for regular simulations and a valuable asset for iterative simulations.

Table 6.1: ASCBR reached after N steps for an L/R=10 pipe with 1000 side facets and varying ASCBR parameters. Here $|\Delta|$ denotes the absolute difference between the analytical and the simulated mean. The batch size was 100000 desorbed particles per data point.

Rel. Error	$\|\Delta\|$	Analytic val.	Sim. mean	ASCBR stop at N_{hit}
1e-5	0.000006	0.190941	0.190935	326.1M
1e-5	0.000003	0.190941	0.190944	346.4M
1e-5	0.000008	0.190941	0.190949	451.9M
1e-4	0.000011	0.190941	0.190952	50.8M
1e-4	0.000031	0.190941	0.190910	69.8M
1e-4	0.000009	0.190941	0.190932	65.3M

7 Advanced Ray Tracing techniques

As we have explored in the preceding chapters, ray tracing is a critical component for Molflow's simulation engine. While chapter 2.3.1 laid the foundation by introducing general concepts of ray tracing, this chapter delves into advanced techniques that address some of the method's inherent complexities, particularly focusing on optimization through Acceleration Data Structures (ADS).

One of the most effective methods to speed up ray tracing queries is the usage of ADS. An ADS is used to partition the search space (here in 3D) in several subpartitions associated with the geometric primitives (polygons or triangles). This significantly reduces the amount of intersection tests that need to be carried out. This is because the algorithm no longer needs to individually test every primitive in the geometry, but rather only those within the relevant partitions.

A unique challenge in optimizing ray tracing for Molflow arises from its specific operational context: while the geometries within Molflow are static, the rays traversing these geometries are not uniformly distributed, strictly adhering to Molflow's distinct Monte Carlo simulation model, which was elaborated in chapter 2.2.3.

Building upon these fundamentals, this chapter aims to introduce innovative methods and algorithms that significantly guide the efficiency of ray tracing routines. We aim to find the ideal ADS that makes use of Molflow's unique requirements. Central to our discussion are Bounding Volume Hierarchies (BVHs) and KD-trees, two prominent classes of ADS. Each offers unique advantages and challenges in their construction and application.

We will first examine the construction algorithms of BVHs and KD-trees. An understanding of these is key in modifying and tuning them to Molflow. The chapter then transitions into an in-depth discussion of splitting techniques used during the construction of these structures. These techniques directly influence the efficiency of ray tracing queries. A significant part of this chapter is dedicated to introducing and evaluating state-of-the-art techniques in this domain. Here, we present our adaptations and innovations, including an adapted version of the Ray Distribution Heuristic and our newly developed Hit Rate Heuristic. The former leverages sample ray distributions to create optimized ADS, while the latter utilizes statistical knowledge from previous simulations to enhance efficiency.

7.1 Construction

The goal of the ADS construction algorithm is to maximise the *quality of an ADS* within a reasonable construction time. The quality of an ADS depends on a particular application and can be related to the total runtime, including both construction and ray tracing time. Software-bound factors are the build time and the efficiency of the traversal algorithm. Construction schemes can put a focus on optimal cache usage during traversal and memory usage. During construction, the quality largely depends on the deployed splitting heuristic and is usually followed by techniques to optimize cache-effects e.g. with coherent memory layouts. Here, we focus on the splitting heuristics. The generally proposed solution is the SURFACE AREA HEURISTIC (SAH). It allows for high quality ADS with minimal construction time. Although not tailored specifically for Molflow's use cases, it still performs very well. Aila, Karras, and Laine (2013) analyse and elaborate the relationship between the SAH and high-quality acceleration data structures, which is why we use it as a baseline for later comparisons.

7.1.1 Bounding Volume Hierarchy

Bounding Volume Hierarchies (BVHs), as we previously introduced in chapter 2.4.1, are a preferred choice for Acceleration Data Structures in ray-tracing, particularly for GPU optimizations. Their efficiency and compatibility with straightforward splitting heuristics, like the Surface Area Heuristic (SAH), make them an industry standard for high-performance ray tracing.

The construction of a BVH in our context is primarily achieved through a recursive algorithm. This method, as outlined in algorithm 5, starts by encapsulating all primitives within a parent node. This node is then recursively subdivided into smaller nodes, a process that continues until certain criteria are met. Although there are other construction schemes for BVHs, such as Linear BVH (Lauterbach et al., 2009) and Treelet Restructuring BVH (Karras and Aila, 2013), our focus on the recursive method is driven by Molflow's specific requirements. The recursive approach adapts well to the spatial distribution of primitives. By analyzing and partitioning the space based on the distribution of these primitives, the recursive algorithm ensures that each node in the hierarchy is optimally structured. In addition, it is flexible in changing the underlying splitting algorithm, making it ideal studying their properties in an attempt to optimize Molflow's ray-tracing engine.

The construction of a BVH using a recursive algorithm is sketched as pseudo code in algorithm 5. The process begins with a parent node encompassing all primitives. This parent node is then recursively subdivided into smaller nodes. The subdivision continues until certain conditions are met, at which point leaf nodes are created. These leaf nodes are fundamental to the structure as they directly contain

Algorithm 5 BVH Construction using RecursiveBuild

```
1: function RecursiveBuild($Primitives$)
2:     $NPrimitives \leftarrow |Primitives|$
3:     if $NPrimitives \leq MaxPrimitives$ then
4:         return CreateLeafNode($Primitives$)
5:     end if
6:     $CentroidBounds \leftarrow$ ComputeCentroidBounds($Primitives$)
7:     $Dim \leftarrow$ ChooseSplitDimension()
8:     if $CentroidBounds$ is degenerate in $Dim$ then
9:         return CreateLeafNode($Primitives$)
10:    else
11:        $Groups \leftarrow$ Split($Dim, Primitives$)
12:        $LeftChild \leftarrow$ RecursiveBuild(Groups[0])
13:        $RightChild \leftarrow$ RecursiveBuild(Groups[1])
14:        return CreateInteriorNode(LeftChild, RightChild)
15:    end if
16: end function
```

the primitives. There are two primary conditions under which a leaf node is created:

1. Primitive Count Threshold: A leaf node is formed when the number of primitives in a node is below a predefined maximum threshold (`MaxPrimitives`). This condition ensures that each leaf node manages a manageable number of primitives, optimizing the efficiency of the hierarchy.

2. Degenerate Bounding Box: A secondary condition for leaf node creation is identified when the bounding box surrounding the primitive centroids becomes degenerate along the chosen split dimension. This occurs when the minimum and maximum coordinates along this dimension are equal, indicating that further subdivision of the node would not be beneficial.

Splitting algorithms differentiate in their selection of the splitting plane. Efficient construction algorithms usually alternate between the x, y, z axes, where optimized algorithms will find the best splitting plane among all three of them. Other approaches that have a lower construction time in mind, alternate the axes based on a criterion, e.g. the axis with the largest extent. This is the approach used in this implementation. The splitting of nodes into two subsets of primitives is guided by a well chosen splitting method. Details on various splitting strategies are discussed in chapter 7.2. These strategies vary based on specific optimization goals. Each split results in two groups of primitives, where one group is used to recursively construct the left child of the BVH, and the other group for the right child. The recursive nature of this algorithm allows for efficient and dynamic construction of the BVH. It adapts to the spatial distribution of the primitives, ensuring that the resulting hierarchy is optimized for subsequent operations like ray-primitive intersection tests. Coupled with an effective splitting strategy, this leads to the creation of a balanced and performant BVH.

In summary, the recursive construction of BVHs, as outlined in this section, offers a flexible and efficient method for further optimizing the ray tracing routine in Molflow effectively, by adapting and enhancing it for the specific requirements of Molflow's simulations.

7.1.2 KD-tree

With properties more relevant to Molflow, we put a particular focus on KD-trees, another form of ADS. KD-trees, as briefly revisited in chapter 2.4.2, present distinct properties compared to BVHs, making them particularly advantageous for static geometries like those in Molflow. One key benefit is their ability to facilitate early ray termination due to the traversal algorithm's preference for nearest candidates, a feature not inherent to BVHs due to potential node overlap following object instead of space partitioning. As illustrated in figure 7.1, the spatial partitioning within the KD-tree demonstrates how different levels of subdivision contribute to the final data structure.

The construction of a KD-tree, similar to a BVH, typically follows a recursive approach. Our algorithm, detailed in algorithm 6, spatially partitions primitives into a hierarchical tree structure.

Starting with an initial set of primitives, the algorithm recursively splits them into smaller groups. A leaf node is created under three conditions:

1. Primitive Count Threshold: When the number of primitives in the current node is less than or equal to a predefined maximum (`MaxPrims`).

2. Maximum Depth: When the recursion depth reaches a threshold, this condition prevents the tree from becoming excessively deep, which could affect both memory usage and query performance.

3. Infeasible split: If a split is deemed infeasible (see line 6, e.g. if it doesn't improve spatial partitioning or leads to unbalanced divisions, a leaf node is favorable.

The splitting function decides how to split the current set of primitives into two groups. Several splitting techniques are discussed in chapter 7.2. After a feasible split is determined, the set of primitives is divided into two groups. Each group is then passed to a new recursive call. This subdivision process continues until the conditions for leaf node creation are met.

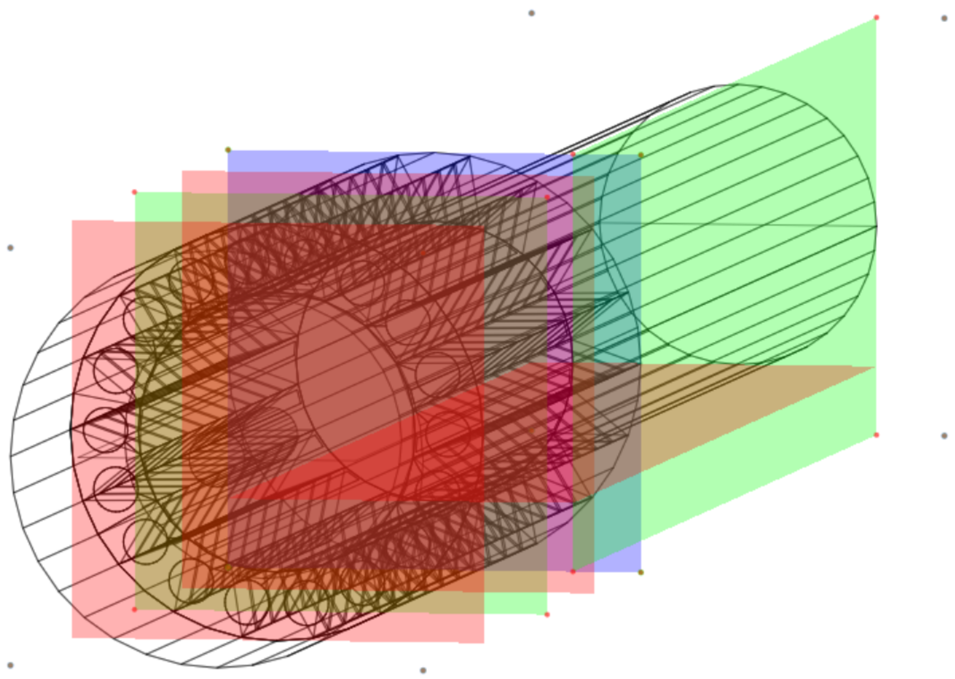


Figure 7.1: Visualization of a KD-tree for an exemplary geometry, illustrating the spatial partitioning and levels of subdivision, colored starting with red, green, and blue.

The resulting KD-tree is particularly suited for ray tracing applications. It allows for efficient traversal and querying, enabling rapid determination of which primitives a ray intersects.

Li, Deng, and Gu (2017) proposed an algorithm for KD-trees that selects splitting candidates with Morton Code and orders primitives on a Morton curve. A similar approach has been proposed by Lauterbach et al. (2009) for BVHs. Although the approach itself is less intuitive compared to conventional greedy-based algorithms, such as splitting by SAH, the authors note that their acceleration data structure achieved "dramatically [shortened] construction times" and high traversal efficiency. Wald and Havran (2006) analyze KD-tree construction algorithms and propose an algorithm using so-called perfect splits. The idea of perfect splits is to clip those primitives that straddle the splitting plane. This solves the inherent properties following KD-trees, that primitives may be referenced by multiple sub nodes. A concept only vaguely following conventional KD-trees is described by Gribble and Naveros (2013). Here, KD-trees are first constructed with a classic algorithm, where only the leaf nodes are kept to create a graph, where each node is connected to their six spatial neighbor relations.

Algorithm 6 KD-Tree Construction for Ray Tracing

```
1: function BuildKdTree($Primitives$)
2:     if $NPrimitives \leq MaxPrims$ or $Depth \geq MaxDepth$ then
3:         return CreateLeafNode($Primitives$)
4:     end if
5:     $Groups \leftarrow$ DetermineSplit($Primitives$)
6:     if Split is not feasible then
7:         return CreateLeafNode($NodeNum$, $PrimNums$, $NPrimitives$)
8:     end if
9:     BuildKdTree($Group[0]$)
10:    BuildKdTree($Group[1]$)
11:    return $Node(LeftChild, RightChild)$
12: end function
```

In a first step, we base our work on the implementation of the recursive construction algorithm by Pharr, Jakob, and Humphreys (2017), which follows the concept of algorithm 6. The recursive construction algorithm is easy to adapt for other splitting heuristics.

7.1.3 KD-tree Traversal

In chapter 2.4.2 we briefly discussed the standard techniques to traverse KD-trees for ray tracing. The `kd-backtrace` algorithm is implemented by Pharr, Jakob, and Humphreys (2017), which serves as our standard implementation. Molflow is a use case, where the spatial awareness of KD-trees is in particular useful. Upon reflection, we recognized the ability to identify the nodes or leaf where the intersection point lies. An idea is to use this position as a start to quickly restart the KD-tree traversal from this position. Further, we believe this is useful for Molflow, as we are tracing particles in an enclosed space. Subsequent reflection points are on average very close to each other.

Popov et al. (2007) described a technique, where KD-trees are adjusted with `ropes`. These ropes are links between bottom-level nodes, enabling space-conscious traversal. The rope traversal algorithm is sketched in figure 7.2 and works as follows. Nodes have direct links to their spatial neighbour nodes. Starting traversal from top to bottom, when a leaf node is reached and no intersection is found, the algorithm utilizes these "ropes" to move directly to an adjacent node at the same level instead of traversing back up the tree. This way unnecessary nodes can be skipped to reduce the traversal steps needed to explore adjacent regions. Other methods would require retracing steps back to the root or higher-level nodes.

We implemented an adjusted traversal algorithm KD Rope Restart will that

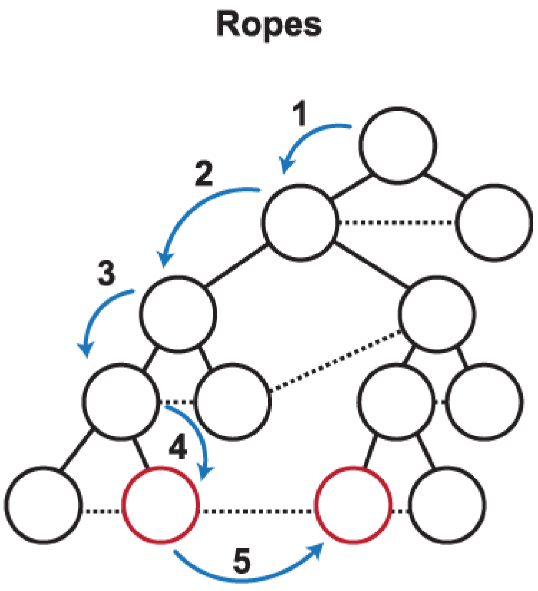


Figure 7.2: KD-tree Rope traversal (Lira dos Santos, Teichrieb, and Lindoso, 2014): The traversal starts from the root node to the bottom. With step 4 we reach a leaf node. Instead of moving back up the tree, we can use a "rope" to move directly to the adjacent node (step 5).

keeps track of the node, containing the facet from the last hit location. In addition to neighbouring links, nodes have direct links to their corresponding parent node. When a terminating hit is identified, the corresponding node is cached with the ray. When a new ray tracing query is started, the traversal will start from the cached node. The algorithm will first move upwards the parent nodes following the parent links until the ray origin is spatially inside a node.[1,2] Starting from there, the standard rope traversal algorithm is used. We sketch our algorithm in 7. In the beginning, we try to fetch the cached node with GetStartingNode. Traversal continues from that node normally. In each node, we check whether the ray intersects the left or right child (see line 5). We can do this by evaluating the ray equation (2.50). In this case:

$$\mathbf{p} = \mathbf{o} + t \cdot \mathbf{r}\,, \tag{7.1}$$

where t is the minimal intersection distance with the current node, $\mathbf{o}$ is the ray origin, $\mathbf{r}$ is the ray direction and $\mathbf{p}$ is the intersection point. Now for the split

[1] As big facets are spanned through multiple leaf nodes, a hit location can be found outside of a node with the facets correct hit location.

[2] Techniques exist that split facets together with the tree subspace.

dimension i, if $\mathbf{p}_i < t_i$, then the ray intersects the *left* region, which is typically according to the minimum coordinate in the respective dimension. A new node is cached for a potential restart when we intersect a leaf node. After we computed the intersection with a leaf node, we use rope traversal by fetching the neighbouring node corresponding to the ray direction with `getNeighboringNode`.

A downside of our implementation lies in `GetStartingNode`. Because facets could potentially lie in two nodes, because they are spatially split, this could give us a bad starting node. The default KD-tree does not split primitives during tree construction. Consider two nodes A, B, both containing the primitive f that a ray will intersect for the `ClosestHit`. Now, the actual hit location on f is located in B, but we are checking intersections with A first. In this scenario, we find the intersection while in node A: A is cached. With an updated intersection distance t, we are still checking B, but terminate, since no primitive is closer than f. `GetStartingNode` will try to restart from the wrong node. This is why we move upwards first until we find a parent node that actually contains the ray. We found that this approach led to best results. It could be improved by implementing KD-trees that split primitives during construction.

Havran and Bittner (2007) proposed a similar approach, though for a bottom-up traversal algorithm. The authors' data structure does not allow direct links to the parent nodes by default, instead they sparsely create augmented nodes, where the following sub-nodes only link to the closest augmented node above in the hierarchy. The authors describe that this leads to reduced memory overhead, with comparable performance to stack-based approaches.

7.2 Splitting heuristics

As we progress from discussing the construction algorithms of Bounding Volume Hierarchies and KD-trees in ray tracing, we connect it to splitting heuristics, which play a crucial role in the quality of an Acceleration Data Structure.

Splitting heuristics in tree-based ADS determine how the 3D space and its encompassing primitives are partitioned. This partitioning is crucial because it directly influences the number of intersection tests during ray tracing, a critical factor in computational performance. For Molflow, where geometries are static but rays follow a non-uniform distribution in line with the Monte Carlo model, selecting an appropriate splitting heuristic is especially challenging and vital.

This chapter delves into the underlying principles and practical applications of these heuristics, beginning with an introduction on the general cost function. The general cost function in ADS serves as a framework for evaluating and guiding the splitting process for different heuristics. It balances various factors, such as construction time and traversal efficiency, to optimize the overall performance of the

Algorithm 7 KD Rope Restart traversal algorithm

```
1: function IntersectRopeRestart($ray$)
2:     $node \leftarrow$ GetStartingNode($ray$)          ▷ Get initial node from $ray$ cache
3:     while $node \neq null$ do
4:         while $node$ is not a leaf do
5:             if $ray$ intersects with $node$'s splitting plane on left then
6:                 $node \leftarrow node.leftChild$
7:             else
8:                 $node \leftarrow node.rightChild$
9:             end if
10:        end while
11:        if $node$ is a leaf and intersects with $ray$ then
12:            $ray.restart \leftarrow node$            ▷ Save restart position $ray.restart$
13:            IntersectLeafNode($ray$, $node$)
14:        end if
15:        $node \leftarrow node.getNeighboringNode(ray)$          ▷ Rope traversal
16:        if $node = null$ then
17:            break                          ▷ Exit if no more neighboring nodes
18:        end if
19:    end while
20:    return $hit$
21: end function

22: function GetStartingNode($ray$)
23:    if $ray.restart$ is a cached $node$ then
24:        $node \leftarrow ray.pay.lastNode$
           ▷ Verify that ray origin is physically inside node
25:        while $node \neq null$ and $ray$ not in $node.bbox$ do
26:            $node \leftarrow node.parent$                    ▷ Move upwards
27:        end while
28:        if $node = null$ then
29:            $node \leftarrow root$        ▷ Fallback to root if no valid node is found
30:        end if
31:    else
32:        $node \leftarrow root$                               ▷ Start at root
33:    end if
34:    return $node$
35: end function
```

ray tracing system. Following this, we will explore various metrics used in splitting heuristics, starting with the Surface Area Heuristic (SAH). SAH is a common choice in many ray tracing applications, due to its adaptability and efficiency. However, in the specific context of Molflow, where ray trajectories are probabilistically determined, the application of SAH and other heuristics may need careful evaluations or adaptation. We introduce two statistical heuristics: The test Ray Distribution Heuristic (RDH) and the Hit Rate Heuristic (HRH). An adapted implementation of the RDH, a heuristic that utilizes a sample ray distribution. The RDH makes use of the fact, that Molflow's ray distribution is non-uniform and dependent on the outgassing ratio of the geometry's facets. In addition, we propose the Hit Rate Heuristic (HRH), specifically designed to capitalize on Molflow's intrinsic statistical data. We will examine how these heuristics operate and their impact on the simulation performance. Further, we will discuss the use of binning strategies in detail to simplify and streamline the splitting process, an important technique in reducing the computational demands of constructing ADS.

7.2.1 General Cost Function for ADS Splitting Heuristics

To build high quality ADS, nodes are usually created recursively in a top-down fashion using a greedy approach. For each iteration, a cost function is evaluated for each splitting candidate to find the locally optimal choice to create new sub nodes. It is typically represented as $C_{A,B}$. This function serves as a tool in estimating the computational efficiency of various ADS configurations, where the goal is to minimize the attributed costs. The general form of the general cost function can be expressed as:

$$C_{A,B} = c_{traversal} + c_{intersection} \cdot (p_A \cdot N_A + p_B \cdot N_B), \tag{7.2}$$

where (A) and (B) represent the left and right child nodes of a split, respectively. Each component plays a specific role in the overall calculation:

- **Traversal Cost** $c_{traversal}$: The traversal cost represents the computational effort required to traverse a node within the ADS. It is a predetermined constant, reflecting the cost of the decision-making process at each node regarding the subsequent path of traversal in the data structure.

- **Intersection Cost** $c_{intersection}$: The intersection cost quantifies the computational load of conducting an intersection test between a ray and the primitives contained within a node. This cost is a fixed constant, approximating the resource intensity of calculating intersections with the geometric primitives in the node.

- **Probability of Intersection** p_A, p_B: The probabilities p_A and p_B represent the likelihood of a ray intersecting with the primitives in the respective left (A) and right (B) child nodes of a split. These probabilities are crucial as they directly influence the frequency of intersection tests, which are central to the efficiency of the ray tracing process.

- **Number of Primitives** N_A, N_B: N_A and N_B denote the number of primitives contained in the left and right child nodes, respectively. These counts are used to scale the intersection costs in proportion to the quantity of primitives in each partitioned space.

The cost function connects the traversal cost with the intersection costs for both child nodes. The intersection cost for each node is derived by multiplying the probability of a ray intersecting that node, the number of primitives in the node, and the fixed cost per intersection test. The primary goal in employing this cost function is to identify the spatial partitioning strategy that minimizes $C_{A,B}$. Lower values of the cost function are indicative of more efficient ray tracing configurations, suggesting reduced computational demands for determining ray-primitive intersections.

The challenge in utilizing this cost function effectively lies in accurately estimating p_A and p_B, which is commonly achieved using splitting heuristics. Accurately estimating the probabilities is not trivial, as it requires a nuanced understanding of the ray behavior within the space. The heuristic must take into account various factors such as the density and distribution of primitives, the nature of ray origins and trajectories, and any specific characteristics of the simulation environment, such as those found in Molflow.

Following the approach suggested in (Pharr, Jakob, and Humphreys, 2017), in our implementation for Molflow, we have set $c_{intersection}$ to 1, as it is typically the more computationally intensive operation, involving multiple virtual function calls. The traversal cost $c_{traversal}$ is set to 1/8, based on the assumption that traversing a node in the ADS is relatively less expensive than performing an intersection test. This ratio of costs reflects the relative computational demands of these two operations in the ray tracing process. In our experiments, it proved to be a good estimation for the costs, which we compared to an automatic evaluation.

In the following sections, we will explore specific splitting heuristics and how they approach the task of determining p_A and p_B in the context of ADS optimization for ray tracing applications.

7.2.2 Surface Area Heuristic (SAH)

The Surface Area Heuristic (MacDonald and K. S. Booth, 1990) is the quasi-standard method for the construction of high quality ADS. It helps to predict the most efficient way to divide space by considering the surface areas of subvolumes. Even though it makes several assumptions that usually don't apply to every use case, the heuristic still delivers very good results with minimal construction time, making it a common choice for many applications.

Figure 7.3 shows two scenarios and different approaches for finding an good splitting position. Here, two scenarios are described, where the first can be solved sufficiently with a split in the center. The second scene shows overlapping and large bounding boxes for the created sub nodes, which is considered to be computationally more expensive than for smaller, non-overlapping boxes. Based on this fact, SAH will find the optimal splitting position for this scenario as show in subfigure (c).

The SAH uses the cost function (7.2) with the following definition of the probability function:

$$p_{i,SAH} = \frac{SA(i)}{SA(V_i)}\,, \tag{7.3}$$

where $SA(\circ)$ denotes the surface area of a volume, which we can simply obtain by computing the sum of all its face areas. Further i denotes an arbitrary sub volume i, where V_i relates to the parent node.

Naturally, the SAH is not the perfect splitting heuristic, even though it delivers on average very good results. The SAH makes simplified assumptions that the rays are uniformly distributed in both their origin and direction. Especially for Molflow's use cases this does not apply. Rays usually start from small surfaces inside a geometry, where the trajectory always relates to a facet's normal. A splitting heuristic that considers this particular trait is the so called Ray Distribution Heuristic (RDH).

7.2.3 Ray Distribution Heuristic (RDH)

The Ray Distribution Heuristic (Bittner and Havran, 2009) is an intuitive technique especially in those scenarios, where rays are not uniformly distributed. This is the case in Molflow, where the origins and trajectories of rays largely depend on the geometry. By utilizing a subset of real rays, the RDH aims to optimizes the construction of Acceleration Data Structures by analyzing how rays are distributed across the 3D space for a given model.

Given a sample distribution of N rays, the RDH is defined using the following probability function:

$$p_{i,RDH} = \frac{|R(i)|}{|R(V_i)|}\,, \tag{7.4}$$

where $R(\circ)$ gives the amount of rays that intersect a node i or it's parent node V_i. Taking into account a fixed initial sample distribution, it is obvious that with each split $R(\circ)$ will become smaller. This is an issue because when the local sample size is too small, it can lead to a statistical bias.

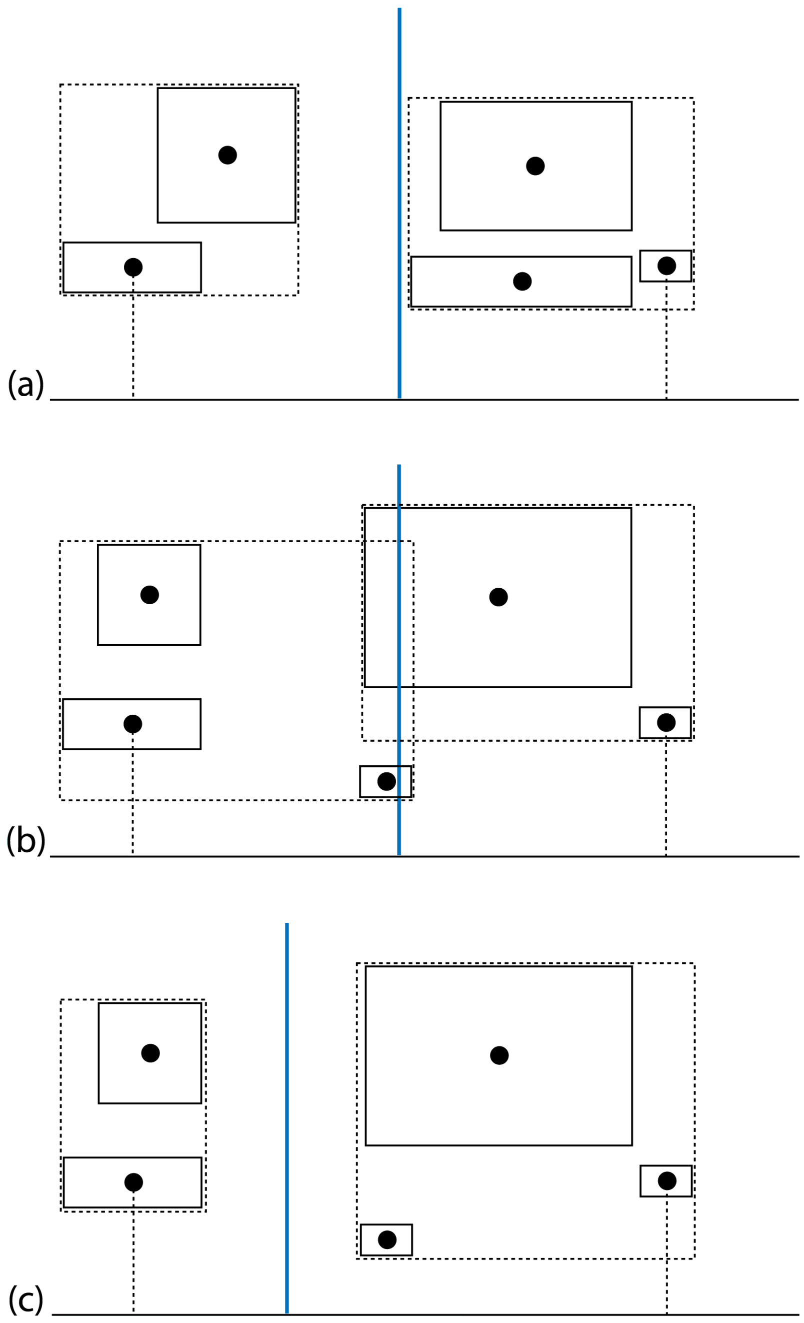


Figure 7.3: Finding the optimal splitting position isn't always straightforward. In (a) a split in the center is sufficient. For the scene in (b) the same splitting position leads to overlapping boxes associated with more computation steps involved. An optimal splitting position for the same scene is shown in (c) (from Pharr, Jakob, and Humphreys, 2017).

The RDH, compared to the SAH, is computationally demanding as for each splitting candidate, two nodes have to be traced with the sample set. Techniques for efficient handling of the RDH during the build process have been proposed by Nabata et al. (2013), such as ray filtering. Considering that for a node only a subset $\mathbf{r} \in R$ of all rays R actually intersect the created subnode, it makes sense to filter only those active rays and use them when evaluating the RDH for further splitting. This is sketched in figure 7.4. Nabata et al. (2013) optimise the algorithm further by minimising potential memory overhead by coupling ray filtering with a technique called skip filtering. Here, rays are only filtered to create a smaller subset, when the difference between the sample set and the rays that did not intersect with a node is sufficiently large. They propose to skip creating a smaller subset when $\alpha > 0.5$, where α is the intersection rate of the current sample set with the new child node.

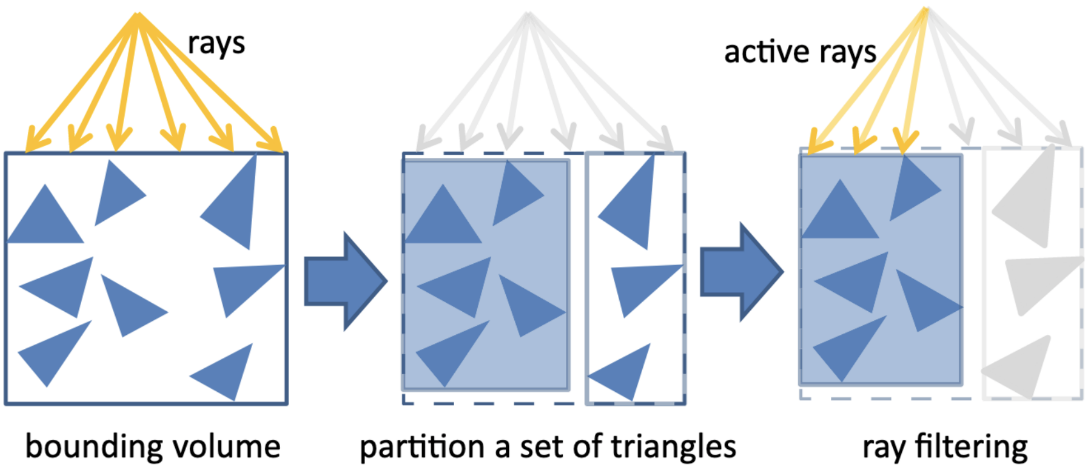


Figure 7.4: Sketch of the RDH. Only a subset of the ray sample intersects a partition.(Nabata et al., 2013)

Shadow Ray Distribution Heuristic (SRDH)

To make up for some of the practical disadvantages associated with the RDH, Feltman, Lee, and Fatahalian (2012) proposed a derived heuristic called Shadow Ray Distribution Heuristic (SRDH). Here, a secondary BVH is constructed solely for the use of *shadow rays*. In classical ray tracing, shadow rays are secondary rays that can be created after a collision with a surface. The authors argue, that shadow rays for graphical rendering applications are much more suitable for the RDH than primary rays. In Molflow we can't directly apply the concept of shadow rays, as such rays have a limited life span unlike the particles traced until they exit a geometry, usually after several thousands of reflections. Essentially, we have to treat each reflected ray as a primary ray.

RDH Sample Distribution

There are various ways to retrieve a sample distribution. The easiest method would be the recording of $\mathbf{R}_{total,k}$ rays and later on to randomly pick a subset of $\mathbf{R}_{\partial} \subseteq \mathbf{R}_{total}(k)$ rays. $\mathbf{R}_{total}(k)$ gives the amount of rays that intersect node k. Considering that $\mathbf{R}_{\partial}$ is sufficiently large, the retrieved subset should give a good distribution, while reducing the computational burden. Nabata et al. (2013) propose that from a given sample distribution only a sample size $\mathbf{R}_{\partial}$ of at most 100 to 1000 rays is sufficient for their implementation of the RDH. In their experiments, they analysed geometries with 75K and 7.9M triangles. They use a fixed sample size $\mathbf{R}_{\partial} = 100$ for $\mathbf{R}_{total}(k) > 1000$, otherwise they use $\mathbf{R}_{total}(k)$ directly to avoid computational overhead of sampling the rays.

One could argue how a good sample distribution can be retrieved. For simplicity, N rays forming a sample distribution can be sampled randomly from a previous simulation. This reduces the bias in the sampling process but might add bias during the splitting process. Especially for smaller sample distributions, the chance that some facets are overrepresented and others are underrepresented by the sample distribution might be stronger pronounced.

A sample distribution that accounts for this effect giving good approximation for the total distribution of rays can be achieved as follows. Given a previous simulation, determine the probability f_i that a ray originates from a facet i either via desorption or reflection with

$$f_i = \frac{H_i + D_i - A_i}{H_{total} + D_{total} - A_{total}} \,. \tag{7.5}$$

Here, H, D and A denote the number of hits, desorptions and absorptions respectively. Indices i and *total* denote the respective numbers for facet i or the *total* numbers of the simulation. Then, given a target sample size of $\mathbf{R}_{total}$ rays, for each facet i we choose r_i rays with

$$r_i = \lceil f_i * \mathbf{R}_{total} \rceil \,. \tag{7.6}$$

Using the ceiling function guarantees, that each statistically relevant facet is represented by at least one sample ray. This might give a sample set of size $\mathbf{R}_{approx}$ slightly larger than proposed, which can be corrected for by removing $\mathbf{R}_{approx} - \mathbf{R}_{total}$ rays from the facets represented by the largest number of rays. While this approach may potentially change the distribution, we found in some initial experiments that it led to slightly better results than the pure random distribution. For an initial study, we follow the guideline to compute splits with $\mathbf{R}_{\partial} = [100, 1000]$ rays at a time, where it is evident, that this recommendation might not work sufficiently well on all geometries. This is, because $\mathbf{R}_\partial$ might quickly become biased and not represent the current sub-section of the geometry well.

7.2.4 Hit Rate Heuristic

An alternative to the Ray Distribution Heuristic is the Hit Rate Heuristic (HRH), which utilizes the empirically evaluated hit ratio of the individual facets contained within a node. Instead of relying on a sampled ray distribution, the HRH uses the results of a previous simulation run to derive the chances of each facet getting hit by a ray.

Given the data from a previous simulation run, a splitting heuristic can be defined with

$$p_{i,HRH} = \frac{|H_{hit}(i)|}{|H_{hit}(V_i)|}\,, \tag{7.7}$$

where $H_{hit}(\circ)$ gives the sum of MC hits on all facets inside a node i or it's parent node V_i. Thus, $p_{i,HRH}$ gives a probability that a facet that is part of node i is hit. The advantage of HRH is that a priori information can easily be obtained independent of the ADS that will be created using the heuristic. This makes it intuitive to deploy in the context of batched simulations (e.g. iterative simulations in chapter 6) as the necessary information for the construction is inherently given after an initial run. The HRH is an intuitive approximation, especially for leaf nodes, as the hit ratio of an internal node typically varies from the corresponding primitive ratios.

As HRH depends on simulation results from a previous run, the construction undergoes two steps. First, we create a sample structure to generate the necessary a priori information. This Progenitor Structure (PS) should be constructed fast with good runtimes, making a BVH constructed with SAH an ideal choice. The goal is to minimize the time it takes to collect $H_{hit}(\Omega)$ MC hits. When the generated $H_{hit}(\Omega)$ is sufficiently large, the data can be used in order to rebuild the PS. We can neglect the additional overhead that arises with the use of the PS, if the total simulation time is significantly larger than the time used for constructing and sampling with the progenitor structure. Our benchmarks still account for construction and simulation time from both the PS and the HRH constructed BVH or kd-tree.

The HRH was designed with iterative simulations in mind (see Chapter 6). Intuitively, the first iteration would deploy a progenitor structure for the whole run. The following iterations have access to the simulation results automatically and can therefore build an adapted acceleration data structure with the HRH. Practically, the relative hit counters of each facet change with each iteration due to the change in simulation parameters. This behaviour emphasises the theoretical downside of the HRH. The goal of a good splitting criterion that tries to optimise the cost function (7.2), lies in giving a good estimate for p_i, the probability of a ray intersecting a node i. The HRH assumes, that a Monte Carlo hit relates to a ray tracing intersection.

A real ray tracing query typically intersects with multiple facets, where the MC hit is only the closest intersection. The average number of real intersections for each ray is given by the scene depth complexity and was elaborate and given for various geometries in chapter 4.

We derived another heuristic from the HRH, which makes use of the real intersection probability of each facet. This approach accounts for intersection tests, that are rejected due to other facets being in closer proximit to the ray's origin. It is less intuitive, as the statistics are not automatically generated with a simulation. Additional counters or an additional pre-processing step, where a sample ray distribution is used to approximate this data, have to be used. In our experiments, we found that HRH, as initially defined, and the modified version give similar results. This is likely because the actual hit probabilities approximate the non-uniform ray distributions following a Molflow simulation sufficiently.

7.2.5 Hybrid heuristics

Certain splitting heuristics may lead to locally biased results when statistical data are limited or when the design of a heuristic inherently does not guarantee good splits. In the case of the RDH (Bittner and Havran, 2009), a hybrid heuristic is suggested to offset potential biases by weighting it against the Surface Area Heuristic (SAH). The authors propose a weighted function that depends on the amount of statistical data:

$$w_r = \alpha \cdot \left(1 - \frac{1}{1 + \beta \cdot |\mathbf{R}|}\right) . \tag{7.8}$$

Here, two constant shape variables α and β are used, with the weight dynamically adjusted solely by the amount of rays $|\mathbf{R}|$. Lower $|\mathbf{R}|$ suggest a stronger bias for certain facets, where a bias for SAH is only inherent, when it doesn't describe the ray distribution properly in general. A hybrid probability can then be calculated by linearly interpolating the RDH and SAH:

$$p_{i,Hybrid} = w_r \cdot p_{i,RDH} + (1 - w_r) \cdot p_{i,SAH} . \tag{7.9}$$

In figure 7.5 the weight function w_r is shown for $\alpha = 0.9$ and $\beta = 0.1$. Using these parameters, the weight of the RDH decreases rapidly for $|\mathbf{R}| \leq 100$.

The hybrid heuristic serves as an effective strategy to mitigate potential biases inherent in heuristics like the RDH, especially when faced with a limited sample set of rays impacting a specific node. A significant challenge in this approach lies in determining an adequately large initial sample set that accurately represents the spatial distribution of rays within subpartitions, up to a certain depth. This is particularly crucial given the substantial computational overhead of RDH in comparison to more traditional heuristics such as the SAH.

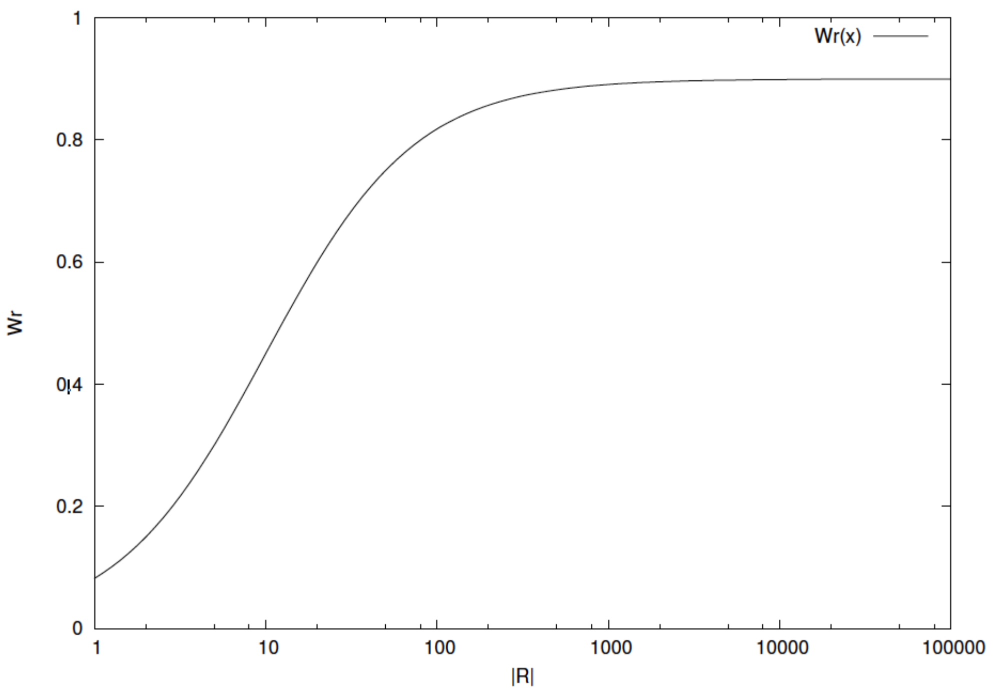


Figure 7.5: Proposed weight function w_r for $\alpha = 0.9$ and $\beta = 0.1$, scaling with the amount of rays $|\mathbf{R}|$ intersecting a node (see Bittner and Havran, 2009).

7.2.6 Binning

Fast construction algorithms typically adopt a greedy strategy, simplifying the complex task of finding the optimal division for tree construction, which is acknowledged as NP-hard (Havran, 2000). These strategies often involve reducing the pool of potential splitting candidates. Intuitively, it makes sense to only consider the object bounds as potential splitting candidates, leading to $\sim 2 \cdot P$ candidates, where P is the number of primitives. However, for larger scenes or when employing more complex heuristics, it's beneficial to further reduce this number to lessen computational demands. Here, the binning strategies come into play.

For binning, instead of evaluating every possible location as a splitting candidate, only N positions are evaluated. Usually, the chosen splitting axis is divided in N equidistant bins for min-max binning (see Shevtsov, Soupikov, and Kapustin, 2007). Here two separate bins are utilised as counters, the min-bin and the max-bin. The min-bin accumulates the primitives whose min-coordinate of the AABB lies inside, respectively for the max-bin. A more general approach simply keeps track of the primitives whose centre coordinate lies inside of a bin interval. Alternatively to spatial binning, one can use an object binning approach, where only the centre of a primitive's bounding box is used as a splitting candidate as opposed to utilising the full bounding box. Instead of equally diving the bins spatially, we can distribute the primitives equally. For N bins and P primitives, each bin keeps track of $\sim P/N$ primitives. This is particularly useful, when considering geometries such as the electron gun from the ELENA experiment, shown in figure 4.1. Here, considering the z-axis, which defines the long side, the distribution of primitives is heavily focused towards the first half. In such scenarios, it would result in $\sim N/2$ bins (bins diving the right half along the z-axis), which are underutilized.

7.2.7 Discussion

This chapter's exploration of various splitting heuristics for KD-trees and BVHs in ray tracing applications, particularly within the context of Molflow, underscores the critical role these heuristics play in optimizing ADS. We want to briefly discuss the expectations for each splitting heuristic when paired with BVHs or KD-trees to minimize the general cost function (7.2) based on their properties.

While most ray tracers build KD-trees using the Surface Area Heuristic (SAH) due to its adaptability and general efficiency, our discussion has highlighted that the unique requirements of Molflow might benefit from deploying different approaches. The non-uniform distribution of rays in Molflow, primarily influenced by the individual geometries, presents unique challenges that standard heuristics may not fully address. The Ray Distribution Heuristic (RDH), initially proposed for BVHs, shows promise in better aligning with KD-tree properties due to its spatial awareness. However, the heuristic's effectiveness depends on the accurate representation of ray paths, which can be challenging given the complexities inherent in Molflow's simulations. The Hit Rate Heuristic (HRH), on the other hand, leverages statistical data intrinsic to Molflow's simulations, offering an innovative approach to tailor the ADS to specific geometries. While it estimates the hit probabilities of individual facets directly, it neglects that ray tracing is steered first through the interior nodes of an ADS. This is where it might create significant overlaps between interior nodes in BVHs without further modifications. Furthermore, the discussion on hybrid heuristics, particularly the combination of RDH and SAH, illustrates a strategic approach to balancing different heuristic strengths. This adaptability is crucial in managing an unintended bias that might negatively impact construction.

Lastly, the binning strategy is key to simplifying and streamlining ADS construction. By reducing the computational load in selecting splitting candidates, these strategies contribute significantly to the efficiency of ADS construction. In our experiments, this was especially noteworthy for the RDH. Even with the limitation to a smaller sample size during each step, the overhead to test all possible splitting candidates was tremendous, making it largely infeasible without a binning approach.

7.3 Evaluation and Benchmarks

In this chapter, we provide examples that highlight the differences between the proposed splitting heuristics used for construction algorithms. This will provide a better understanding of the potential effects in particular between the hybrid heuristic and non-hybrid heuristics. The effectiveness of the different techniques is evaluated with benchmarks running on a set of test cases with varying properties. Splitting criteria are evaluated for both BVH and KD-trees. For KD-trees we further distinguish between different traversal techniques.

7.3.1 Hybrid heuristics

The hybrid heuristic implements a method to connect two splitting heuristics with the idea to counteract the downsides of using one for the entire tree construction process. Originally, the hybrid heuristic was proposed because the RDH would lead to biased results for lower levels of the tree, because less rays from the sample ray distribution are relevant for the sub nodes after each split. Here, the SAH acts as a good balance due to its good performance on average.

To visualise how the splitting heuristic lead inherently to different results, we evaluate splitting candidates for SAH, RDH and the hybrid heuristic at different steps during the splitting process. These examples are sketched in figures 7.6, 7.7, 7.8, and 7.9. In each example, the same state during a construction was used to evaluate the heuristic scores across the whole plane. The test cases further consider the impact that a decreasing amount $|R|$ has on the RDH and thus the hybrid heuristic. Figure 7.6 shows a completely different pattern for RDH and SAH, showing that they are fundamentally different. The RDH would have selected a splitting candidate for the minimum on the left border, where the SAH was more oriented towards the center. Since the stage was only using a relatively low amount $|R|$, the hybrid heuristic is weighted more towards the SAH, leading to a splitting candidate in the center that is less ambiguous due to the reduced right shift. The example in figure 7.7 shows similar patterns for SAH and RDH. The RDH has a distinct minimum, whereas the cost function for the SAH is continuing to go slightly downwards all the way towards the right side. The hybrid heuristic with an equal weight is balancing out this effect, hence it selects a candidate similar to this found by the RDH. Figure 7.8 shows an example with fewer split candidates. The curves for RDH and SAH are relatively similar, but still result in different split locations. Here, RDH splits at 60% and SAH at 40%. Another scenario with a lower $|R|$ is given in figure 7.9. Here, the SAH and RDH curves are completely different. RDH identifies better results on the left edge, taking 20% as the minimum. The SAH curve is balanced and found a split in the center at 50%.

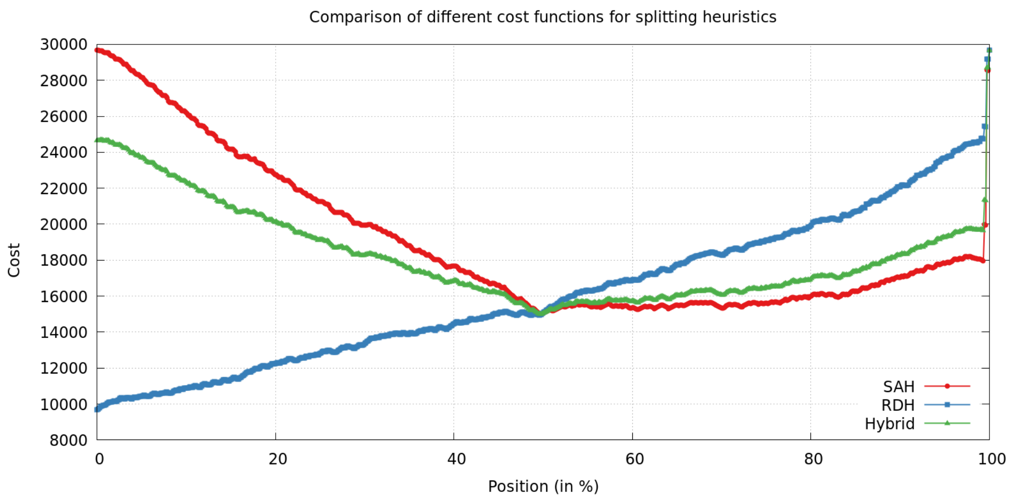


Figure 7.6: X-axis shows the splitting position in %. The Y-axis shows the cost associated with a split at a particular position. The cost function is according to (7.2) for the corresponding splitting heuristic. RDH identifies a minimal position on the at the left boundary. SAH finds a splitting candidate on the right half of the node. The hybrid method finds a clear candidate in the center.

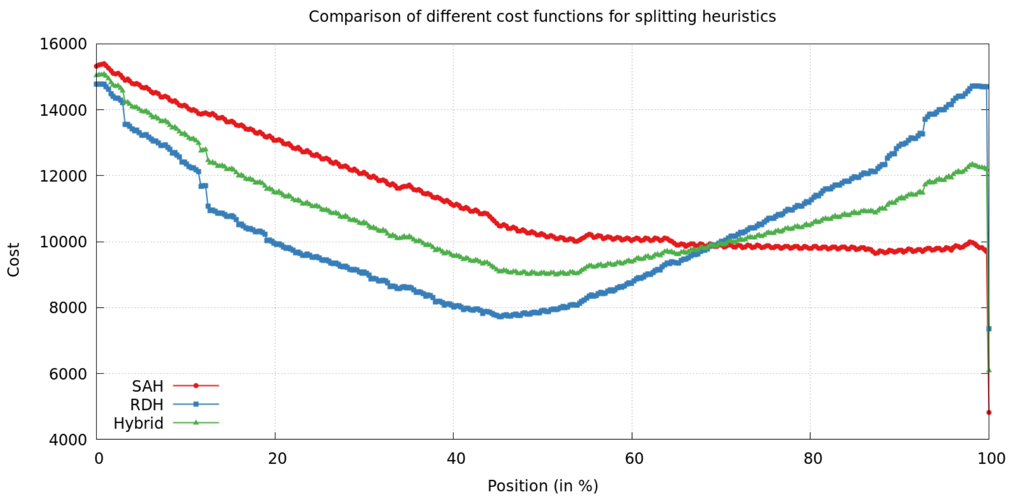


Figure 7.7: X-axis shows the splitting position in %. The Y-axis shows the cost associated with a split at a particular position. The cost function is according to (7.2) for the corresponding splitting heuristic. An example with fewer spit candidates. RDH identifies a candidate close to the center. SAH finds a splitting candidate on the right boundary.

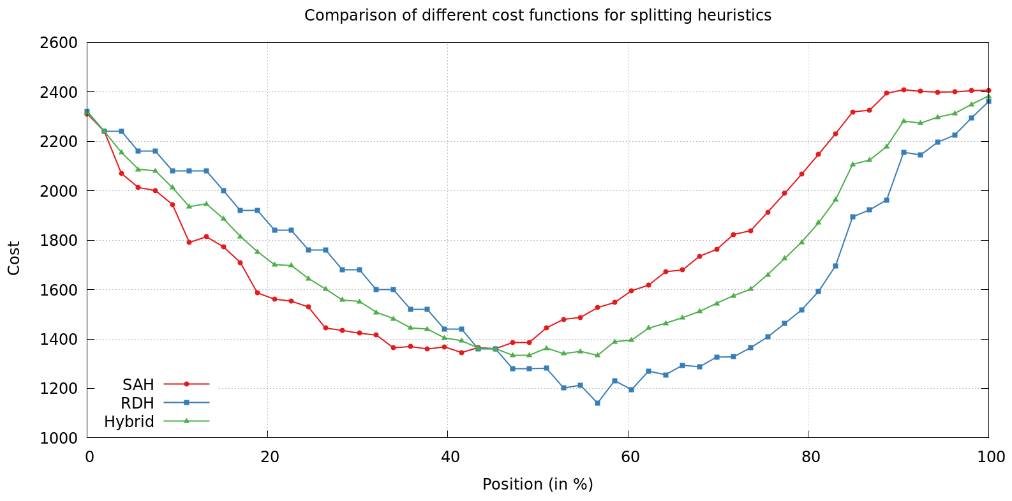


Figure 7.8: X-axis shows the splitting position in %. The Y-axis shows the cost associated with a split at a particular position. The cost function is according to (7.2) for the corresponding splitting heuristic. RDH identifies a candidate close to the 60% position. SAH finds a splitting candidate on the 40% mark.

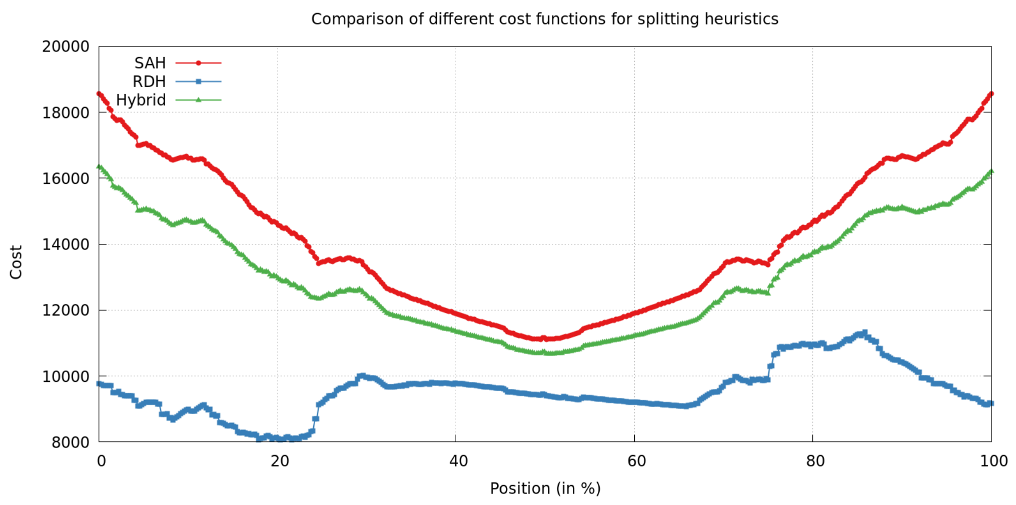


Figure 7.9: Example #4 for splitting cost functions. Here, the RDH leads to a significantly different pattern than the SAH, which resulted from a small test ray sample size. This resulted in the same minimum for both the SAH and the hybrid approach.

7.3.2 Benchmark

This chapter elaborates and discusses the benchmarks on the various implemented techniques surrounding splitting techniques and traversal algorithms that have been implemented for BVH and KD-trees. In figure 7.10 we show in a box and whisker plot, how the different techniques perform compared to the BVH with SAH splitting as the default technique. On average, the BVH with RDH performs well and has slightly better performance on average than the default procedure. The KD trees are worse on average. It is noteworthy though that most techniques perform well in some particular test cases noted as outliers. The KD tree using SAH proves to be a good default technique for the SAH, where the hybrid approach for the KD tree leads to the most performant individual benchmarks of the whole set. The rope traversal techniques did not excel. The newly developed rope restart technique is slightly better than the original implementation. We would like to argue, that the technique shows potential. In particular, when the traversal technique and the underlying data structure are further optimised. We found, that one of the downsides for the rope traversal, are cases, where facets belong to multiple nodes. This can happen, because a KD tree is dividing primitives spatially, where primitives always belong to one or another node at a split for a BVH. A technique that could reduce this effect, is object splitting. Primitives that would belong two both nodes after a split, will be cut at the splitting axis and divided into two new polygons, one belonging to each node.

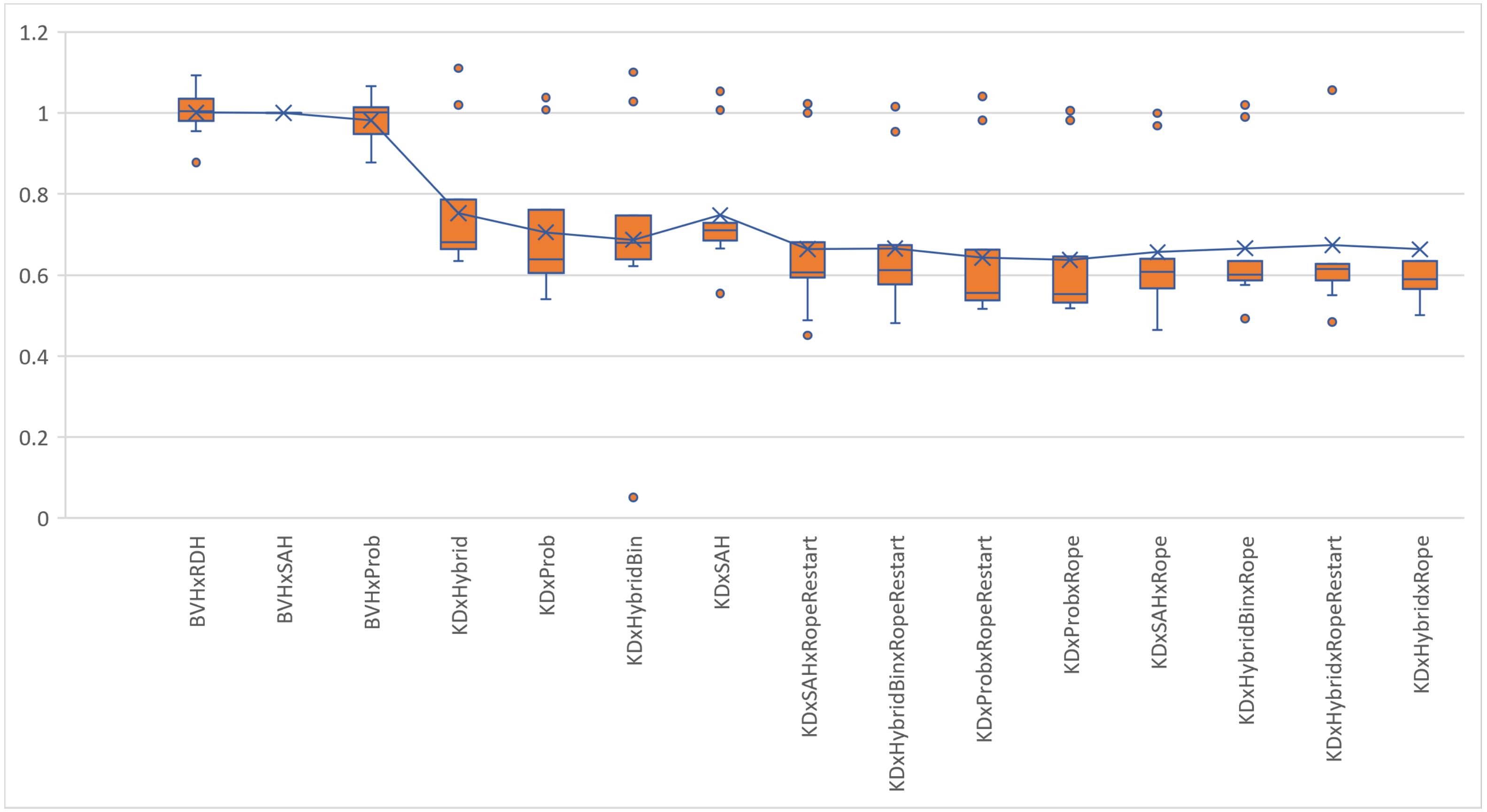


Figure 7.10: Different ADS and varying splitting techniques and for KD-tree also rope traversal has been benchmarked for a set of test cases. Y-axis gives the relative performance (MC hit/s) compared to BVH with SAH splitting (BVHxSAH). Each box incorporates half of the data points of all benchmarks. The line inside the box shows the median. The dots outside of the boxes represent strict outliers. BVH with SAH and RDH metric are the most performant on average. KD-tree implementation performs well only in certain test cases.

7.4 Outlook

We discussed and evaluated various combinations of techniques for both the BVH and KD-tree acceleration data structures for ray tracing queries. Research focus recently shifted more towards BVHs, but for special cases, such as Molflow, KD trees might still bring some advantages. In a first attempt to integrate KD trees in Molflow, the average performance proved to be a step backwards compared to the new baseline that was set with the BVH using SAH. In a few special cases the KD tree, particularly with the hybrid applying RDH and SAH, was able to outperform the BVHs. These positive outliers of the KD tree performance show, that an appropriate acceleration data structure could be chosen dynamically, e.g. depending on the geometry properties. Overall, our KD-tree implementation still has potential in various forms for optimisations, e.g. in terms of better caching effects. For BVHs, various such techniques such as the use of treelets instead of linear nodes were already researched by Pantaleoni and Luebke (2010), but they do not translate well to KD-trees.

We further presented an optimised adaption of the RDH for KD-trees, by identifying the pros and cons of similar techniques such as the SRDH, that creates a separate ADS for primary and shadow rays in graphical rendering applications. The splitting technique has promising traits for Molflow but ultimately led to a lower quality ADS compared to the SAH in most cases. The KD-tree implementation with rope traversal only showed decent results on average. An ambiguous restart position leads to an additional computational overhead, as the cached starting position is counterproductive. This happens when a primitive is part of multiple leaf nodes. A hit can then find the closest intersection point on a particular primitive, but the actual collision location might be – spatially – inside another node, which should be the actual restart node. We only gave a brief look into the potential that an adaption of the KD-tree has. It shares several of the traits that are theoretically fitting for Molflow's ray tracing engine. It is left unclear though, whether performance improvements can be expected, when the underlying base, the KD-tree, is already subpar.

8 GPGPU Kernel

General-Purpose Computing on Graphics Processing Units (GPGPU) harnesses the computational power of Graphics Processing Units (GPUs), traditionally used for rendering graphics, for a wide range of applications beyond their original scope. GPU computing is widely used for scientific applications e.g. for data analysis or data reduction in High Energy Physics (HEP), for example by Schouten, DeAbreu, and Stelzer (2015) or Bruch (2020).

Molflow users typically use the software in various stages when they are designing vacuum components or to run simulations for a case study. In the initial development phases, consumer-grade hardware is commonly used. However, as the need for high precision arises, more powerful remote resources are employed. GPUs are widely used in scientific fields to accelerate all kinds of simulations in the HPC domain. A GPGPU simulation kernel would enable Molflow users to rapidly evaluate their designs using hardware that they likely already own, especially when complex geometries or time-demanding simulations are involved. Based on NVIDIA's OptiX API for ray-tracing queries, a C++/CUDA kernel has been developed.

First, we introduce some of the basics concepts of GPU programming with CUDA. Further, we elaborate on the topic of ray tracing on GPUs and the connection of the OptiX API to Molflow. This leads to the discussion of the development and evaluation of the implementation of a straightforward port of Molflow's ray tracing engine and compares how it performs on both GTX and RTX hardware, where the latter introduced hardware-accelerated ray tracing on consumer-grade GPUs. Lastly, we introduce the kernel that was optimised for the use of RTX hardware and elaborate the design, where an in-development version has already been discussed in detail by Bähr et al. (2022). Furthermore, we focused on accuracy-related components.

8.1 GPU Basics

To understand the design choices and guidelines for the development of the proposed GPU kernel, it is important to become familiar with some of the concepts and terminology used in the context of GPUs and GPU programming. Specifically, we focus on the CUDA toolkit (Cook, 2012) and the NVIDIA hardware architecture. Following a brief introduction to the main concepts, we introduce some best known practices, that are key to developing performant GPU applications.

A critical difference to note between GPUs and CPUs is their preference for data precision. GPUs are commonly optimized for single-precision floating-point computations (FP32), whereas CPUs are typically more geared towards double-precision computations (FP64). CPUs, with their capacity for FP64, are better suited for tasks demanding higher numerical precision. Whereas GPUs excel with their throughput on FP32 operations. In scientific simulations such as Molflow, it is essential to strike a balance where numerical precision is sufficient to ensure the accuracy of the computations without adversely affecting performance. This makes it a key concern, which we validate in later chapters.

8.1.1 Threads

In CUDA, threads are the fundamental units of computation, distinguished by their ability to execute parts of a parallel algorithm independently. Unlike CPU threads, which are significantly less following the Multiple Instructions Multiple Data (MIMD) paradigm, CUDA threads are designed for fine-grained data processing, operating on individual elements of large data sets following the SIMD (Single Instruction Multiple Data) model. Each thread has access to its local memory for independent data manipulation. Furthermore, threads are logically grouped into CUDA blocks and further into a grid, where both threads and groups indexing happens with 3D variables for natural division into vectors, matrices and volumes. Threads within the same block can share data efficiently using the shared memory space provided by the block, facilitating intra-block communication and synchronization.

8.1.2 Blocks

A block in CUDA is a collection of threads that are scheduled to run on the same Streaming Multiprocessor (SM). The block structure allows threads to cooperate on a given task and share resources effectively. All threads in a block have access to a shared memory space, which is faster than global memory but limited in size. This shared memory is vital for threads within the same block to communicate and synchronize their operations, making it possible to perform complex tasks that require inter-thread coordination.

8.1.3 Warps

Within a CUDA block, threads are organized into warps, which are the basic execution units. A warp consists of a group of threads, typically 32, that execute the same instruction simultaneously but on different data, adhering to the Single

Instruction, Multiple Threads (SIMT) model. This architecture allows for efficient parallel processing but can lead to inefficiencies in cases of thread divergence, where threads of the same warp need to execute different instructions. Each block is divided into multiple warps, and the efficient management of these warps is crucial for maximizing the performance of a CUDA kernel.

8.1.4 Streaming Multiprocessors

The Streaming Multiprocessors (SM) are the core of NVIDIA's GPU architecture, responsible for executing CUDA blocks. Each SM can execute multiple blocks simultaneously, depending on the resource requirements of the blocks. SMs include warp schedulers that manage the execution of warps within the blocks. These schedulers are designed to optimize the concurrent execution of warps, ensuring high throughput and efficient utilization of the GPU's computational resources.

It is important to understand the differences between the SIMT architecture used in CUDA and the traditional SIMD paradigm, as they fundamentally differ in how they handle multiple data streams:

- **SIMT:** In CUDA's SIMT architecture, each thread in a warp can execute the same instruction on different data. This model allows individual threads to follow different execution paths (branching), which comes at the cost of potential efficiency loss due to thread divergence. SIMT is designed to maximize the utilization of GPU cores by enabling more flexible and granular control over parallel computations.

- **SIMD:** Traditional SIMD architectures, commonly found in vector processors and some CPUs, also execute the same instruction across multiple data points simultaneously. However, SIMD lacks the flexibility of SIMT in handling diverse execution paths. All elements in a SIMD operation must execute the same instruction sequence without branching, making it less adaptable to complex, conditional data processing scenarios.

While SIMT offers greater flexibility and is well-suited for complex, conditional operations common in diverse computing tasks, SIMD excels in scenarios where the same operation is uniformly applied across large data sets. SIMD remains an important architecture better reflecting the diverse and dynamic nature of current and emerging computational challenges.

This architectural design is visualized in Figure 8.1, which illustrates the relationship between individual threads, thread blocks, warps, and their mapping onto the physical architecture of a GPU, including CUDA cores and Streaming Multiprocessors, and the structure of a complete GPU.

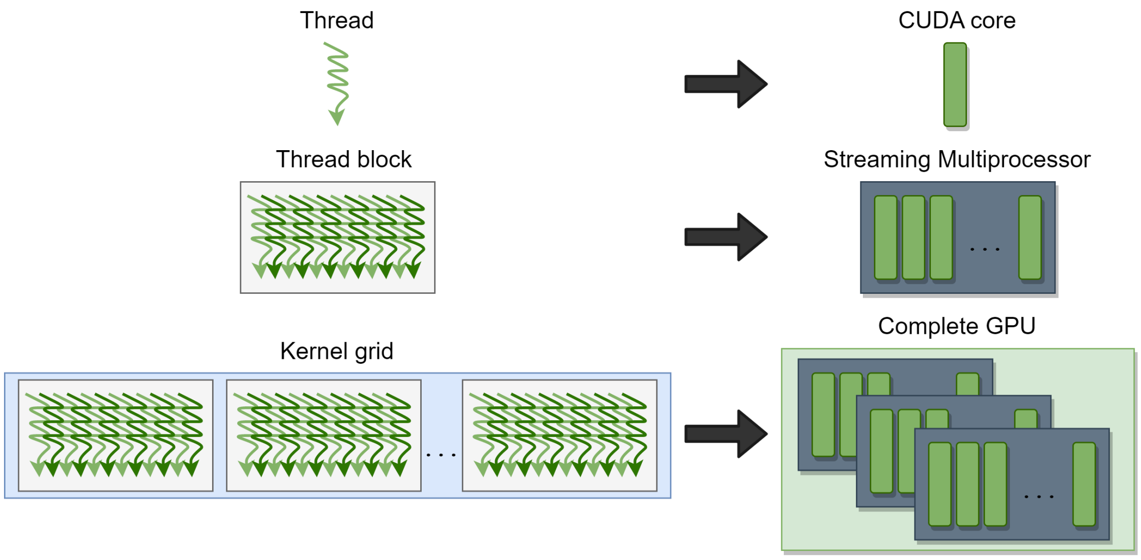


Figure 8.1: Hierarchical organization of threads, blocks, and grids in the CUDA model and their correspondence to GPU hardware components (cf. NVIDIA, 2023).

Following this overview of the computational architecture, we will now examine how the individual elements interact with the GPU's memory hierarchy to achieve high-performance computing in GPGPU applications.

8.1.5 Memory Space

The efficient execution of CUDA kernels not only depends on the optimal organization of threads but also depends on how they interact with different types of memory available on the GPU. In this section, we delve into the various memory spaces within the CUDA environment, including host memory, device memory, and global memory. We will discuss how each memory type is accessed and utilized by threads, blocks, and warps, highlighting the significance of memory management in achieving optimal performance in GPU-based computations.

Host memory: Offloading code to the GPU requires control by a host, usually a CPU. The host memory is not directly linked to the GPU device. Accessing host memory from the device and vice versa requires expensive memory transfers. Hence, where possible, data should be saved in the GPU's own device memory for frequent accesses. Device memory might pose a limitation and needs to be considered when designing an application enabled for GPU computing. This is especially true for consumer-grade GPUs: e.g. NVIDIA RTX 3060 has around 8GB memory[1]. On the other hand, there are data center GPUs e.g. NVIDIA H100 with 80GB memory[2]. Such powerful GPUs are not commonly available to all engineers though.

Device memory: Device memory can be addressed in multiple fashions, which largely depends on the GPU architecture. Important to know to optimise for performance is whether the memory is located on-chip or off-chip, where the former offers faster access and is therefore more desirable. For NVIDIA GPUs with CUDA the memory hierarchy usually resolves into the following for on-chip memory. We refer to NVIDIA's TU102 architecture, which is sketched in figure 8.2. Each thread has its own local registers and memory. Furthermore, each streaming multiprocessor contains *shared memory* that can be accessed by all of its threads of the SM and is shared between thread blocks and used for interthread communication. Depending on the architecture, there may be L1 caches per multiprocessor and shared L2 caches between multiprocessors. The caches are used to keep frequently or recently accessed data in available for fast reads.

Global memory: In CUDA, *global memory* refers to the device memory that is accessible by all threads across all streaming processors, as well as the host CPU. This memory is primarily used for memory allocation and memory copy operations between the host and the device. It's the largest memory space available on the GPU and is suitable for storing data that needs to be accessed or modified by multiple threads or cannot fit into faster, on-chip memory types. However, it is important to note that global memory has relatively high latency and lower bandwidth compared to on-chip memories like shared memory. Therefore, optimal use of global memory is crucial for maximizing performance. Techniques like coalesced memory accesses, where threads access contiguous memory locations, can significantly improve bandwidth utilization.

Additionally, while global memory is generally off-chip and slower, certain specialized memories map to global memory but have unique characteristics:

- **Texture Memory**: This is a read-only memory optimized for texture fetch operations. Texture memory uses a dedicated cache (L1 cache), which can be beneficial for certain types of irregular memory access patterns.
- **Constant Memory**: Another form of read-only memory. It is used for variables that do not change over the course of a kernel execution and are shared across all threads. This memory type has its own cache (constant cache) on

[1]Checked 17/11/2023: `https://www.nvidia.com/en-us/geforce/graphics-cards/30-series/rtx-3060-3060ti/`

[2]Checked 17/11/2023: `https://www.nvidia.com/en-us/data-center/h100/`

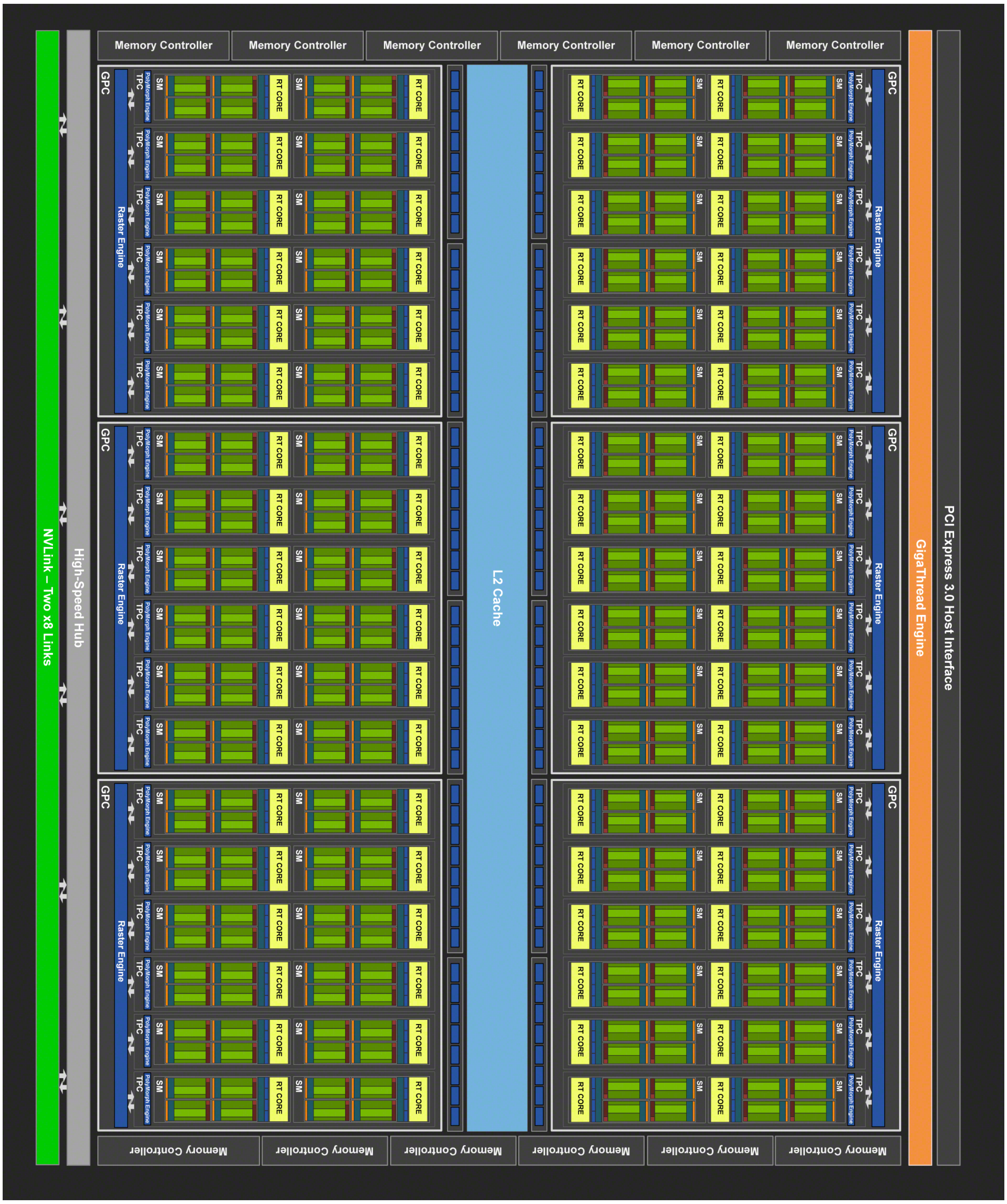


Figure 8.2: TU102 architecture.

the chip, providing efficient access when the same constant data is used by multiple threads.

While global memory is versatile and essential for large-scale data handling in CUDA, its effective use requires careful consideration of access patterns.

8.1.6 Cooperation and Synchronisation

In CUDA, effective collaboration between threads within a block is central to optimizing performance. Threads can share data either through the slower global memory or the much faster, on-chip shared memory. The choice between these two types of memory impacts both the speed and efficiency of data exchange among threads.

Especially when threads within a block are writing to and reading from shared memory, thread synchronization instructions are critical. CUDA provides explicit synchronization instructions (e.g. `__syncthreads()`) for this purpose, ensuring that all threads within a block reach a certain point in the code before any of them proceed. This synchronization is crucial for maintaining data integrity and avoiding race conditions.

The size of a block influences how threads can cooperate. Larger block sizes allow more threads to work together, potentially improving data sharing and throughput. However, this also increases the time needed for block synchronization. Additionally, larger blocks can affect the occupancy of a Streaming Multiprocessor (SM). The occupancy is the ratio of active warps to the maximum number of warps supported on a SM. As more warps within a block reach the synchronization barrier, the SM's occupancy temporarily decreases, potentially impacting its ability to hide latency.

As mentioned in NVIDIA's CUDA programming model (Cook, 2012), threads within a warp (a group of 32 threads) are implicitly synchronized after each instruction. This means that threads in a warp execute instructions in lockstep, meaning that each thread performs the same instruction simultaneously. Developers can leverage this feature to avoid the need for explicit synchronization in scenarios where problems can be effectively divided among warps. However, implicit synchronization in warps has its drawbacks as it strongly depends on a particular GPU architecture, e.g. due to leveraging particular warp sizes.

8.1.7 Best Practices in CUDA Programming

Writing performant CUDA kernels requires careful consideration of the underlying hardware and the software requirements. The goal is to design kernels that *minimize thread divergence*, that is an effect of conditional branching within warps.

Thread divergence can lead to performance degradation as divergent paths are serialized within a warp. Understanding and effectively utilizing the different types of memory (shared, global, constant, and texture memory) is important to *optimize memory usage*. Where possible, data access should be speeded up by minimising data transfer between host and device and by using shared memory. However, it is important to note that extensive use of synchronisation mechanisms, particularly with shared memory, can significantly impact performance. Another goal is to optimise the number of threads per block and the blocks per SM to *maximize occupancy* and fully utilise the hardware. To calculate the occupancy, an occupancy calculator can be used to find the best configuration. It offers a quick and theoretical way to optimize kernel configurations for maximizing the use of CUDA cores. Higher occupancy generally means better utilization of the GPU resources. However, it does not always guarantee the best performance. A combination of using the occupancy calculator for a baseline and tuning via empirical testing usually leads to good results. *Warp-level primitives* exist for efficient data sharing and computation within a warp, reducing the need for synchronization and shared memory. *Profiling* the application via NVIDIA Nsight[3] is a crucial step in optimizing GPU kernels further. By identifying bottlenecks and focusing on both computational efficiency and memory access patterns, the application can be optimized accordingly.

8.2 Ray tracing on GPUs

It's especially due to NVIDIA's introduction of Turing-based GPUs in 2018 that ray tracing has become feasible again for real-time rendering in graphically demanding scenes. According to NVIDIA (2018), Turing GPUs allow for a major speedup in Giga Rays per second compared to Pascal GPUs from the previous generation. For a GTX 1080 Ti, about 1.1 GRays/s could be achieved, while RTX 2080 Ti achieved about 10 GRays/s in various benchmarks. Today, dedicated ray tracing units are integrated in most consumer-grade GPUs, e.g., NVIDIA RTX series, AMD RX series, or Intel Arc series, as well as data center GPUs, e.g., NVIDIA H100 or Intel Data Center GPU Max 1550. Exemplary for NVIDIA RTX, developers have access to hardware-accelerated ray traversal and ray-primitive intersection algorithms.

Ray-tracing algorithms are often described as *embarrassingly parallel*, meaning they can be easily divided into independent tasks with little need for inter-task communication. Intuitively it makes sense to leverage the capabilities of graphical processing units to maximise the simulation performance for Molflow and Synrad's simulation kernel.

As Molflow's performance is heavily dependent on the Ray Tracing algorithm, trying to benefit from the advances in hardware-accelerated ray tracing is a logical

[3]A profiling tool part of the CUDA toolkit.

step. One has to consider if the usage of modern GPUs, in general, can be beneficial compared to CPUs, or if the benefits – if any – will only be feasible with new generation GPUs, which come with hardware acceleration for Ray Tracing. Previous comparisons between GPU and CPU performance conducted by Roberto Kersevan and Pons (2009) showed negligible gains from using GPUs, because “the execution diverges quickly depending on the successes or failures of the intersection tests (’if-then-else’ branches in the code)”.

In this chapter, we give a brief introduction to general ray tracing algorithms for the GPU, further introduce the commonly used ray tracing APIs and elaborate on the potential problems for the design of a GPU kernel for vacuum simulations with the usage of the introduced RTX GPUs.

8.2.1 Hardware accelerated RT

Optimizing for NVIDIA Turing’s architecture and using the RT cores to the utmost capabilities requires adapting Molflow’s ray tracing algorithm to conform with their hardware accelerating functions. Specifically, these RT cores “accelerate Bounding Volume Hierarchy (BVH) traversal and ray/triangle intersection testing (ray casting) functions” (NVIDIA, 2018, p.30).

Molflow traditionally employs BVHs (refer to chapter 2.4.1), which theoretically aligns well with the structural capabilities of RT cores. However, the practical implementation of adapting these BVHs to the RT cores requires careful consideration, especially in terms of compatibility and performance optimization.

A critical aspect of this adaptation is the geometric representation of facets. While currently described by polygons, an evaluation is necessary to determine if converting these to triangles – a format native to RT cores – would indeed offer a performance boost. This tessellation process should consider not only the potential speedup but also the accuracy and integrity of the results.

The introduction of Turing-based GPUs has been a game-changer for ray tracing, providing not only the raw computational power but also specialized hardware designed to handle ray tracing tasks efficiently. For NVIDIA GPUs, the NVIDIA’s OptiX API abstracts the complexities of directly managing GPU resources for ray tracing.

8.2.2 GPU ray tracing with OptiX 7

With the advent of such powerful GPUs, APIs like NVIDIA’s OptiX 7 offer a streamlined approach to harnessing this power. OptiX provides a higher-level interface for ray tracing, simplifying the process of implementing ray tracing algorithms by managing the underlying CUDA calls and RT core utilization. This allows developers to focus on the ray tracing logic itself, rather than the intricacies of GPU programming.

OptiX is particularly well-suited for scientific applications, as it is designed to work seamlessly with NVIDIA's GPU architecture, from older Maxwell generations to the latest Turing-based models. It enables the efficient generation and tracing of rays using the GPU's parallel processing capabilities, which is ideal for the independent nature of particles in simulations like Molflow.

In principle, Molflow+'s ray tracing algorithm is well suited for the Single Instruction Multiple Data (SIMD) architecture on GPUs. Given the independent nature of particles in molecular free flow, one can maximize the parallel workload by generating and tracing individual particles per thread. Our focus is on exploiting the RT cores in NVIDIA's recent RTX hardware, as detailed in the Turing architecture white paper (NVIDIA, 2018, p. 30).

8.2.3 OptiX Basics

According to the OptiX Programming Guide (see NVIDIA, 2020) we describe several technical terms connecting the API to previous discussions for ray tracing (see chapter 2.3.1).

A full ray-tracing kernel in OptiX consists of a ray generation kernel, a ray-primitive intersection test using the RTX pipeline and kernels for the trace processing:

- **Ray Generation:** This is the first stage where rays are generated. Implemented as a CUDA kernel, this stage sets up the initial parameters of rays such as origin, direction, and energy. Afterwards, the ray follows the RTX pipeline to find the intersection point.

- **RTX Pipeline:** The RTX pipeline refers to the set of operations that are hardware-accelerated on RTX-capable GPUs. This pipeline includes:
  - **Scheduling:** The scheduling blocks within the RTX pipeline manage the distribution and execution of tasks on the GPU. This stage ensures that the hardware-accelerated operations are efficiently queued and processed.
  - **Acceleration Structure Traversal:** This process involves navigating the geometry's BVH to find potential hit points for the rays.
  - **Intersection:** This step computes whether a ray intersects with any geometry in the scene.

- **Trace Processing:** After a ray has potentially intersected with an object in the scene, the trace processing stage determines the result of that intersection. This stage is divided into:

  - **Closest Hit:** If an intersection is confirmed, the closest hit program – a CUDA kernel – computes the effects that apply to the interaction between the ray and the intersected object.
  - **Miss:** If no intersection is found, the miss program – a CUDA kernel – is called. For Molflow this can be used to track particle leaks that occur when a particle exits the bounds of a vacuum component without getting physically *pumped*.

- **Secondary Ray Launch:** This optional step, allows for the generation of new rays from the point of intersection, which is used for recursive computations, skipping the ray generation step.

Figure 8.3 shows a sketch for the OptiX 7 pipeline. The green labels highlight the parts, that can make use of proprietary algorithms, which are hardware-accelerated on compatible GPUs. First a custom *ray generation* kernel is called. From here ray tracing is launched using the RTX pipeline on hardware. ADS traversal always utilizes the built-in method, which is hardware accelerated. A built-in intersection test for triangles is provided for maximum leverage of the RTX hardware. Self-implemented kernels for intersection tests for custom primitive are only software-accelerated. If no intersection is found, a `Miss` kernel is called. Otherwise, the `Closest Hit` kernel is called for processing the object intersection.

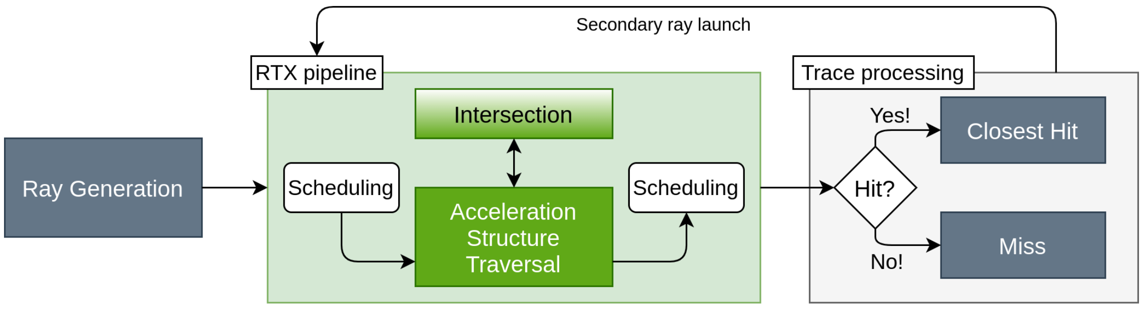


Figure 8.3: Ray tracing pipeline with RTX technology cf. Wald and S. G. Parker, 2019.

This chapter outlines the integration process and key considerations in adapting Molflow's ray tracing algorithm to leverage OptiX 7.

8.2.4 Integration Process

The development of Molflow's GPU simulation kernel has been aligned with NVIDIA OptiX 7.0 and CUDA 10.1, leveraging the hardware-accelerated ray tracing capabilities of NVIDIA GPUs with RT Cores, like the Turing series. This chapter outlines the integration process and key considerations in adapting Molflow's ray tracing algorithm to leverage OptiX 7.

The migration of Molflow's simulation code to OptiX 7 involved several important steps:

1. Initialization: Calling `optixInit()` to initialize the OptiX API and ensuring GPU compatibility.

2. Context Creation: Establishing a CUDA-to-OptiX device context for managing ray tracing operations.

3. Module and Program Group Creation: Compiling CUDA-based ray tracing programs into OptiX modules and organizing them into program groups for different ray types and polygon interactions.

4. Pipeline and SBT Configuration: Assembling a pipeline linking program groups and setting up the Shader Binding Table to dictate shader execution based on ray interactions.

5. Geometry Handling: Loading the geometric data onto the GPU and defining facet properties (e.g., hardness, opacity).

6. Acceleration Structure: Computing the BVH to optimize ray traversal efficiency.

7. Execution: Launching the ray tracing process with customized kernels and handling memory management intricacies, such as managing facet counters and avoiding race conditions.

The integration process begins with initializing the OptiX API to confirm the compatibility of the GPU and the readiness of the API for use. Subsequently, a device context connecting CUDA with OptiX is established, laying the groundwork for all ray tracing operations. Modules, which are compilations of CUDA kernels for the ray tracing tasks (see pipeline in figure 8.3), form the core of OptiX 7's programmability. These tasks are then organized into program groups containing post processing routines that define the interaction of rays with the scene. These interactions span from the generation of rays, their behavior upon missing geometry, to their effects when hitting an object. In classical ray tracing coming from computer graphics, these routines are also called shaders. The Shader Binding Table (SBT) is a critical component that binds these ray tracing shaders to specific geometries and materials in the scene, determining the execution of shaders based on ray interactions. This structure is essential in ensuring the correct routines are invoked for each ray during the trace tracing process. With the input geometry, OptiX creates and manages a Bounding Volume Hierarchy to speed up the ray tracing. The launch command then initiates the ray tracing, defining the number of rays to be processed and setting the rays in motion across the scene's acceleration structures.

8.2.5 Challenges in OptiX integration for Molflow

Integrating NVIDIA's OptiX into Molflow presented several challenges, addressed through advanced features and careful design considerations.

Thread divergence: A challenge, which has already posed problems in a previous attempt to port Molflow's simulation engine to the GPU (Roberto Kersevan and Pons, 2009), is to minimize the effects of thread divergence. This divergence, where post-processing varies by facet type, affects particles' reflection and absorption. Techniques such as teleportation or linked facets from the CPU kernel, as cited by Marton Ady (2016), were not considered for the initial GPU kernel, making it necessary to provide GPU-specific solutions. Modelling different facet behaviors, such as transparent and reflective surfaces, required innovative approaches within the OptiX framework. The Shader Binding Table (SBT), a feature highlighted in the OptiX Programming Guide (NVIDIA, 2020), was instrumental in mapping specific programs to facets, akin to function pointers, reducing the impact of diverging threads. We describe the implementation in more detail here in chapter 8.3.2.5. The SBT, alongside NVIDIA's GPU work creation feature, allowed for more efficient workload handling and ray tracing invocation, a process described by V.V.Sanzharov et al. (2019).

Memory management: Efficient memory management was crucial, especially considering the substantial data requirements for tracking collisions at each facet. Shared memory for thread blocks and data type optimization were strategies employed to address GPU memory limitations (see chapter 8.1.5). Optimal performance also considered different approaches for random number generation using cuRAND (see chapter 8.3.2.10). The decision between iterative versus recursive ray tracing approaches was explored to balance performance and memory requirements. Early tests indicated potential benefits of a recursive approach, which warranted further investigation and optimization (see chapter 8.3.3.3).

Precision: Dealing with precision limitations, especially in single-precision contexts, was essential to avoid computational inaccuracies. This involved implementing strategies for managing potential leaks and miss-traces which we elaborate in great detail in chapter 8.4. Wald (2021) discusses the issues related to numerical precision in intersections programs. He explains where single-precision routines are sufficient and elaborates that in the field of scientific codes double-precision calculations are usually relied on. In figure 8.4 we visualise a problem in Molflow arising from utilizing non-sufficient precision. It shows two scenarios, where in one case the reflection point remains inside the simulation volume and in another it is numerically on the outside of the volume. Depending on the reflection angle this will cause an *unphysical* leak. This is leading to a false negative report due to arithmetical error, as particle is not able to exit the geometry physically in this scenario.

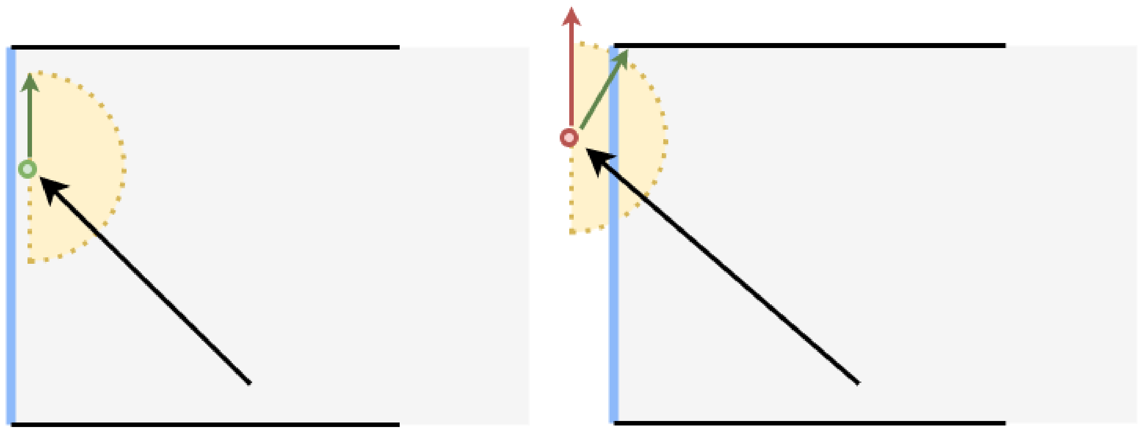


Figure 8.4: Left: A particle reflects at the left surface (blue line). The reflection point is numerically on the inner side of the volume. The yellow half circle depicts the possible reflection angles – oriented in relation to the facet normal. Right: Same scenario, but the reflection point is numerically on the left side of the volume. A reflection angle perpendicular to the facet normal (red arrow) will then lead to a leak, no further reflection point can be found. Though, another angle (green arrow) will lead the particle back into the volume – the reflector facet is "invisible" to the ray tracer.

Primitive type: For maximum leverage of the RTX hardware, we have to use the built-in intersect routine after identifying if it is feasible. For a complete study, we also implement Molflow's standard approach, where a ray-polygon intersection algorithm is utilized, and compare it to the RTX-capable ray-triangle intersection test (see chapter 8.3.2.3).

Ray tracing query: As emphasized in the OptiX Programming Guide (see NVIDIA, 2020) utilizing an `Any Hit` program negatively impacts the performance. In this study, with the goal to maximize the performance benefits of the RTX technology and with the assumption that it is a feasible strategy, we utilize only a `Closest Hit` and a `Miss` program. Utilizing the performant `Closest Hit` kernel while preventing self-intersections and efficiently computing transparent facets posed significant design challenges.

Gribble, Naveros, and Kerzner (2014) explains certain problems that a ray tracer can encounter when continuing from an identified closest hit position, such as when surfaces are deployed that are essentially in contact with each other, depending on the utilized traversal algorithm. Molflow's ray tracing algorithm naturally deploys a hybrid between closest-hit and any-hit traversal. This approach is crucial when using transparent facets for gathering internal statistics, as the algorithm must track facet crossings. Normally, a closest-hit kernel suffices to find reflection points, but complexities arise in practice. For example, when a particle crosses a transparent facet, it doesn't reflect but still provides data. The CPU-based method lists all intersected facets along a ray, using only those up to the reflection point. However, dedicated ray tracing cores mainly support closest-hit kernels for efficiency. Adapting these kernels for transparent hits, by continuing with the same trajectory from a new hit point, can cause issues like self-intersections, which are further discussed in chapter 8.4.

The integration of OptiX 7 into Molflow represents a significant step towards harnessing the power of modern GPU architectures for high-performance, hardware-accelerated ray tracing. This chapter has outlined the critical steps and challenges encountered in this integration, as evidenced by the RTX pipeline visualized in figure 8.3, setting the stage for further advancements and optimizations in Molflow's GPU-based simulation capabilities.

8.3 GPU Kernel development

Fundamentally, Molflow's simulation algorithm is based on ray-tracing, which was always thought to benefit a lot from SIMD processing via GPUs, due to its independent work load. Surprisingly, a previous study about the GPU compatibility for Molflow concluded, that the benefits of using a GPU kernel for the simulations are marginal. Benchmarks only resulted in a 3.3× speedup on the GPU compared to a single-core CPU. This was attributed mainly to the computationally demanding and complex nature of the algorithm. In particular, the large amount of if/else statements lead to thread divergence on the GPU (see Roberto Kersevan and Pons, 2009). Mainly driven by its purpose for 3D computer graphics, such as movie rendering or computer games, there have been constant advancements in developing more efficient algorithms or dedicated computing units, for example Deng et al. (2017). Recent advancements in GPU hardware did not only result in an increase of raw processing power, but also in specially designed ray-tracing units e.g. with NVIDIA's RTX GPUs. With the Turing architecture (see NVIDIA, 2018) – and also its recent successor Ampere – NVIDIA introduced so called RT cores. This hardware unit implements the ray-tracing algorithm directly on the hardware, opposed to the multi-purpose CUDA cores. Comparing an RTX 2080 against its predecessor GTX 1080 TI, based on the Pascal architecture, the new hardware has been claimed to be ~ 10 times faster in terms of processed rays per second for an official ray tracing benchmark.

In this chapter, we revisit the suitability of Molflow's algorithm for GPU processing by investigating the possible performance increase with both a software- and hardware-accelerated GPU ray tracing engine based on NVIDIA's ray-tracing API OptiX 7 (see S. G. Parker et al., 2010). Our approach leverages the capabilities of dedicated ray tracing units as much as possible and tries to work around the hardware-bound constraints. Considering a common vacuum component with well-known properties, we investigate the potential performance gains, as well as the question of whether the reduced precision for the ray tracing algorithm has a negative impact on the precision of the results that would make it an unfeasible choice for simulations with Molflow. For comprehensive details on the Monte Carlo method and ray tracing in Molflow, refer to 2.2.3 and 2.3.1.

The remainder of the GPU related research is organized as follows. First, we consider the contributions of other studies that researched the usability of RTX hardware, with a focus on comparable physical MC simulation applications (see section 8.3.1). In section 8.3.2, we discuss the design choices for Molflow's GPU simulation engine. At last, in section 8.3.3 we conduct a performance and precision study to evaluate how the GPU code compares to Molflow's CPU implementation for different parameters and geometries, where we use a set of cylindrical vacuum tubes with varying length-radius ratios.

8.3.1 Previous work

Studies have already been undertaken on the use of RTX hardware for applications beyond typical rendering or even ray-tracing tasks. For example, Wald, Usher, et al. (2019) applied a point-in-polyhedron picking algorithm to the common ray tracing problem. They achieved speedups of up to $\sim 6.5\times$ with an optimized RTX kernel compared to a full CUDA implementation on a TITAN RTX, while on the predecessor TITAN V without RT cores the CUDA implementation even prevailed.

Molflow is a Monte Carlo simulation software used for vacuum simulations. For similar applications, which are used to study different physical phenomena and are also based on a ray tracing-based engine, some initial studies for the usability of RTX hardware have already been conducted. With an RTX compatible extension for OpenMC (see Romano et al., 2015), speedups of about $\sim 33\times$ could be observed for triangle meshes compared to comparable CPUs (see Salmon and McIntosh-Smith, 2019). OpenMC is a Monte Carlo particle transport simulation code commonly used for modeling of nuclear reactors, which is capable of simulating various nuclear reactions for neutrons and photons. For the open-source software Opticks, an OptiX powered GPU engine is used for the simulation of optical photons as an interface for Geant4. Referencing RTX hardware, the author observes speedups of $\sim 1660\times$ compared to single threaded CPU simulations and a speedup of $\sim 5\times$ for simulations with the RTX feature disabled (see Blyth, 2020). Discrepancies in the results were mainly attributed to be geometry related, where remaining discrepancies were briefly attributed to arithmetic precision. Opticks uses CSG to represent geometries instead of polygon- or triangle-meshes, which are not further accelerated by RTX hardware.

Given that potential performance benefits using RTX acceleration have been studied in comparable Monte Carlo simulators, we give a further outlook for RTX usage in addition to a performance study by including a comparison of the accuracy of our algorithms and discussing the real-life suitability for such simulations with a well-studied vacuum component. Further we elaborate a few design choices, such as the handling of random number generation or the usage of thread local counters, which are necessary for a fast and more robust kernel and which can be applied to similar physical applications.

8.3.2 Molflow's GPU implementation

In this section we present our design for the ray-tracing programs using the OptiX API as well as choices related to the memory and random number generation. We give a brief explanation of the parts that are straightforward ports from the CPU algorithm, which is discussed in greater detail by Marton Ady (2016).

The OptiX API handles the traversal and intersection routines directly with built-in algorithms and utilizes the hardware-acceleration features of NVIDIA RTX GPUs, thus only the ray generation and trace processing procedures are the main concern when trying to achieve a high performance ray tracing algorithm. For a complete study we compare the performance of the built-in ray-triangle-intersection with a port of Molflow's ray-polygon intersection routine. One major restriction with the API is, that the built-in algorithms work only on 32 bit representations of the geometry and the rays. This drawback will be discussed together with the performance and precision study of the kernel (see section 8.3.3). Despite the drawbacks, we decided to focus further on the usage of the hardware-accelerated ray-triangle algorithm, due to its proven speedup compared to software-based solutions.

8.3.2.1 Barycentric coordinates

In ray tracing, when a ray intersects a triangle, the intersection point can be expressed in barycentric coordinates. These coordinates are derived as part of the intersection test, which calculates the intersection point and its relative position within the triangle. This is crucial property for accurately determining hit locations and texture coordinates at the intersection point, as these can be interpolated based on the barycentric coordinates to obtain Cartesian coordinates.

Barycentric coordinates originate from the concept of a centroid in geometry. They can be defined for different types of polygons, where we focus only on their use for triangles. They are a set of three numbers that represent the relative weights of the vertices of a triangle towards a specific point within that triangle. In essence, they indicate how much each vertex contributes to the position of a given point. Given a triangle's vertices $\mathbf{A}, \mathbf{B}, \mathbf{C}$, then a point $\mathbf{p}$ inside this triangle can be expressed by:

$$\mathbf{p} = \alpha \cdot \mathbf{A} + \beta \cdot \mathbf{B} + \gamma \cdot \mathbf{C} \,. \tag{8.1}$$

Here, (α, β, γ) are the barycentric coordinates of $\mathbf{p}$, following the condition:

$$\alpha + \beta + \gamma = 1 \,, \tag{8.2}$$

and $\alpha, \beta, \gamma \geq 0$. Figure 8.5 sketches equation (8.1) to visualise the transformation from barycentric coordinates to Cartesian coordinates for point $\mathbf{p}$, here in 2d coordinates $\mathbf{p} = (u, v)$.

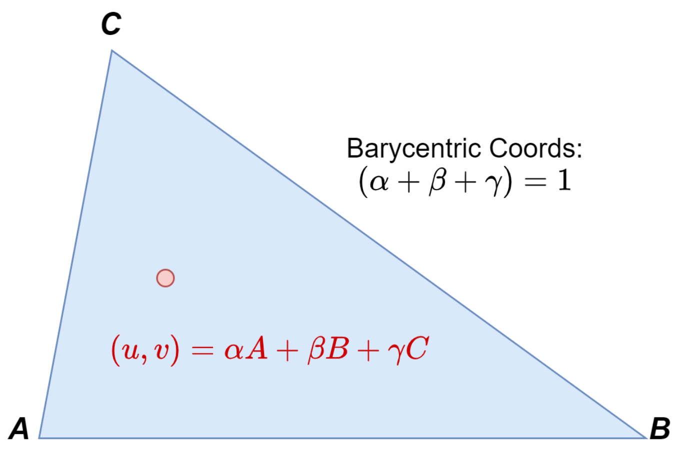

C
Barycentric Coords:
$(\alpha + \beta + \gamma) = 1$
$(u, v) = \alpha A + \beta B + \gamma C$
A
B

Figure 8.5: Visualisation of how barycentric coordinates α, β, γ are used to interpolate a point (u, v) for a set of vertices $\mathbf{A}, \mathbf{B}, \mathbf{C}$.

Each barycentric coordinate represents the proportion of the area of the triangle opposite to the respective vertex, divided by the total area of the triangle. For example, α is the ratio of the area of the triangle formed by $\mathbf{p}$, $\mathbf{B}$,$\mathbf{C}$ to the area of the entire triangle $\mathbf{A}$, $\mathbf{B}$,$\mathbf{C}$. Barycentric coordinates allow transformations in both 2D and 3D coordinate systems, for $\mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{p} \subseteq \mathbb{R}^2$ and $\mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{p} \subseteq \mathbb{R}^3$, respectively.

8.3.2.2 Primitive Types

Acceleration Data Structures (ADS) can be built using various primitive types. These primitive types are tested for intersection when a leaf is reached during the traversal algorithm. Most ray-tracing applications primarily use triangles as their primitive type. Triangles are versatile, as they can represent arbitrary polygons, and their intersection tests are highly suitable for SIMD architectures. Additionally, ray-triangle intersection tests have been a primary focus for research in the ray-tracing domain, making them highly optimized (Wald, 2021).

Molflow supports the importation of STL files, which primarily use triangles to describe CAD geometries. However, Molflow has historically been designed for polygons. Arbitrary polygons are well-suited for depicting planar walls, and their internal handling allows for straightforward optimisations, such as reducing memory load for counting structures. There are other noteworthy primitive types that are not applicable to Molflow's use cases. Notably, RTX GPUs support hardware-based intersection tests for curve and sphere primitives.

8.3.2.3 Polygons vs. Triangles

In Molflow's native implementation, there exists only a general N-polygon ray-tracing kernel. For SIMD architectures, this is not ideal, because intersection tests will always have to compare the incoming ray dynamically with N-points. This is why GPU ray tracers usually convert polygon primitives into triangles. For the initial implementation of Molflow's GPU kernel, we ported the polygon ray-tracing kernels to CUDA to work in place of the native triangle kernels from the OptiX API. Initially, the polygon-based kernels were used to validate the code with the original CPU version. In OptiX (see chapter 8.2.2) the BVH traversal and triangle intersection routines are implemented as part of the proprietary API and can make use of hardware-acceleration. For Molflow, using different primitives does not simply work by calling a different intersection routine. Also, their hit results have to be interpreted differently. Triangle intersection results are returned as barycentric coordinates (see chapter 8.3.2.1) by the API. As polygon's stay as the base for the GUI and more efficient handling of textures, these results have to be mapped back on to a parent polygon.

For barycentrics returned for triangle-intersection tests, this can simply be achieved on the kernel side by interpolation according to equation (8.1). We achieve this performantly by precomputing the 2d texture coordinates, that are the vertices of a textured triangle expressed in local coordinate space. This is analogous to TEXTURE MAPPING techniques used in rendering applications. For Polygons, on the other side, no further modifications are necessary. Here, the intersection test returns hit coordinates directly in local 2d coordinates.

Table 8.1: Benchmark on NVIDIA GTX 970 for the software-accelerated triangle intersection algorithm from OptiX 7.0 and the ported kernel using polygons. Total execution time is shown in seconds for various geometries. Geometry #1 and #3 is the cylindrical tube (L/R=100, with 1000 side facets). Geometry #2 and #4 is the ELENA e-gun geometry (see figure 4.1). In addition, tests #3 and #4 deploy several textured facets to put additional stress on the trace-processing routines.

Primitive/Geometry	#1	#2	#3	#4
Triangle	497.04	964.57	477.87	948.87
Polygon	618.78	1317.03	702.58	1370.69

We compared the feasibility of utilizing triangles instead of polygons in an initial test, using an NVIDIA GTX 970 GPU and OptiX 7.0[4]. This GPU has no dedicated ray tracing units, so the OptiX API falls back to a software-accelerated implementation for triangle intersections. Comparing these triangle intersection tests against a custom implementation for polygons, based on Molflow's original algorithm (see chapter 2.3.3), the triangle implementation proved to be faster in all scenarios. We compared the performance running a simple tube geometry (L/R=100, with 1000 side facets) and the ELENA e-gun geometry (see figure 4.1) without and with additional textures. Here, the triangle implementation was faster by a factor of 1.2× – 1.4× as shown in table 8.1.

Potentially, a hand-coded intersection routine gives the advantage of running application-specific instructions during the traversal steps at the expense of not utilising hardware acceleration. During the initial design phase, we researched the benefit of deploying intersection tests with double precision for polygons. Due to the BVH traversal, which is still done with the OptiX API's in-built algorithm using single-precision, some issues could still not be completely solved. For the rest of the study, we aimed to develop a more robust solution for triangles, as they are significantly faster using hardware-acceleration, which is shown in great detail in chapter 8.3.3. Another benchmark comparing both implementations on the NVIDIA RTX 2060 GPU and the NVIDIA RTX 3090 GPU, provided in table 8.2, shows the native triangle implementation to be faster by a factor of 1.7× – 3.1×.

8.3.2.4 Triangulation methods

Already part of Molflow to add compatibility with the STL (Standard Triangle Language) file format used in most CAD software packages, a triangulation algorithm is

[4]At the beginning of the study, our lab machines did not have a RTX GPU. Hence, in the first steps, we investigated the feasibility of a GPU kernel solely using the OptiX API.

implemented to convert arbitrary polygons into triangles. Molflow uses fan triangulation (see De Loera, Rambau, and Santos, 2010), which does not care about any of the properties of the resulting triangle mesh. As polygon areas can have a large impact on performance and triangles with acute angles can further result in bounding boxes for an ADS that lead to worse performance than when a triangulation method would be used that considers such properties. Various triangulation methods can be employed to convert polygons into triangles. The choice of triangulation method depends on factors such as the desired balance between computational complexity and the quality of triangles. For Molflow, the method has to respect certain properties. The method has to handle both convex and concave polygons with or without holes.

A triangulation method that is commonly preferred in many domains, where triangle meshes are used for simulations, is the Delaunay triangulation. The triangulation method is credited with creating high-quality meshes for several use cases. It leads to uniformly sized triangles, and thus avoids triangles with comparably long or short sides. Compared to fan triangulation, Delaunay triangulation is computationally more demanding. As geometries that are commonly used in Molflow tend to have relatively low amounts of polygons and triangulation is usually applied as a single pre-processing step, it is not expected to lead to a major loss of performance.

We implement Delaunay triangulation (Delaunay et al., 1934) in Molflow with the assumption, that it will lead to better performance by effecting the created acceleration data structures. The implementation uses CGAL (Computational Geometry Algorithms Library)[5] by utilising its data structures and algorithms as an interface for this process. The *iterative Delaunay triangulation* algorithm creates a mesh by applying the Delaunay property. For every triangle in the triangulation, the circumcircle (the circle passing through its three vertices) contains no other input points in its interior. Given a set of initial points (the vertices), a so-called super triangle is created that contains all points. Then, for each point, the algorithm identifies the triangle containing that point. This triangle is subsequently removed from the triangulation, and the point is connected to the vertices of the removed triangle. For each triangle, the Delaunay property is checked. If it is violated, the edges are swapped with those of adjacent triangles to create new triangles. The process is then repeated until each point has been processed. An example for the first two steps is sketched in figure 8.6. Here, in a first step an arbitrarily large super triangle is created that contains all vertices. Next, one of the points connected with the vertices of the containing triangle: the super triangle. This happens first by removing the triangle as a whole and creating three triangles by connecting the super triangle's vertices with the point.

In a small experimental study, we compare the performance for the electron gun

[5]Accessed on 20/08/2023: `https://www.cgal.org/`

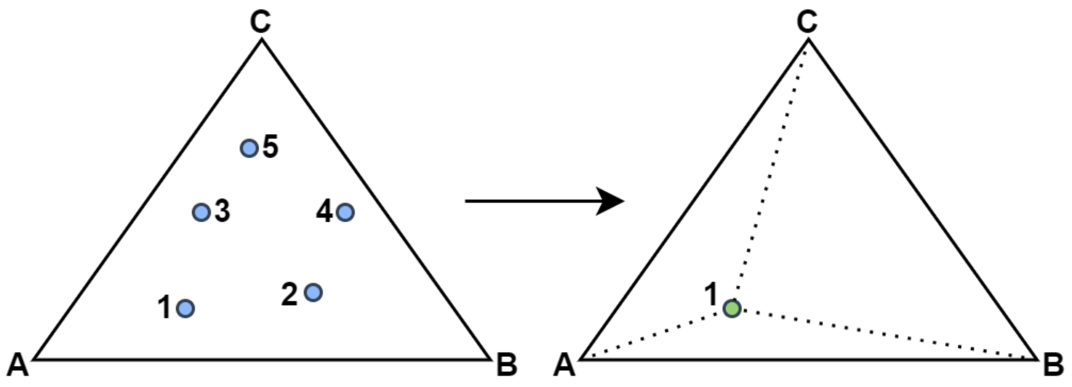


Figure 8.6: Example for a delaunay triangulation. In the first step, all pictures are surrounded by a *super triangle*. In the next step, the vertices of the super triangle are connected with the first point. In practice, the super triangle ABC is the triangle containing point 1, it is then *deleted* to create three new triangles $A1C$, $1BC$, $AB1$.

geometry from the ELENA experiment (see figure 4.1) for both triangulation methods: fan triangulation and Delaunay triangulation. Delaunay triangulation achieved 11.6% speedup using the hardware set with the NVIDIA RTX 3060 compared to fan triangulation. Further, the generated mesh resulted in better accuracy, as less "skinny" triangles had been created, causing problems for the adaptive offset along the normal as previously mentioned. For less complex vacuum tubes such as the cylindrical tubes (see figure 4.6), the improvements were negligible. Side facets are responsible for the biggest chunk of the work. Because they are rectangular, they are divided equally for both algorithms. As we couldn't find any test case, where the performance using Delaunay triangulation has been worse, we strongly recommend this as the new standard method. Achieving better improvements with other triangulation methods might be possible, but we like to argue that Delaunay triangulation is a proven method in many domains and other methods are likely to lead to improvements only in a subset of test cases. Hence, Delaunay serves as a solid base. We like to note, that due to the chronological order in which the experiments were conducted, fan triangulation has been used as the standard method for all corresponding benchmarks. Delaunay triangulation has only been implemented and validated at the end of this study.

8.3.2.5 Triangle types

We want to highlight that the algorithms are deeply affected by the choice of using 1-sided or 2-sided triangles. Rendering applications tend to use face-culling techniques to only render a certain polygon, when it is facing the view point and currently visible. Face-culling can dramatically increase performance, as intersection tests can be quickly discarded. Intuitively, this technique can also be applied to Molflow. Considering a cylindrical vacuum pipe, the only sides that are of interest are those facing inward. Now, if a particle is located on the outer part, e.g. due to numerical error, this could cause undesired effects. If we wouldn't apply back-face culling, a ray would intersect with the closest facet's outward-facing side: they ray can not re-enter, thus, propagating the numerical error. In figure 8.7 examples are sketched for transparent passes and reflective hits.

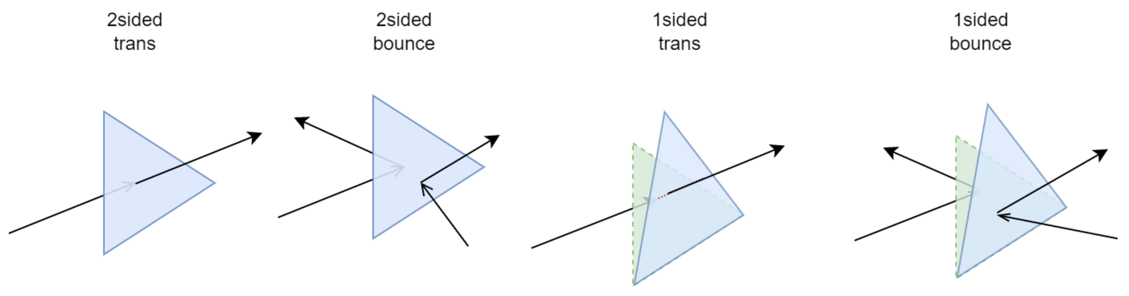


Figure 8.7: How either 2-sided or 1-sided triangles affect the ray tracing algorithm with back-face culling. A 2-sided transparent simply ignores a reflection event, but may allow to gather statistics. A bounce on 2-sided facet can happen on both front and back sides. To replicate the same effect for 1-sided transparent facet, we need to deploy a clone (colored in green) with opposite orientation, otherwise statistics from an intersection through the back-side can not be gathered. For a 1-sided solid facet, a backwards-oriented clone also helps in recreating the correct effects.

The choice between 1-sided and 2-sided triangles can influence the efficiency and accuracy of ray tracing algorithms. One-sided triangles may result in faster intersection tests, as they have only one surface to check for intersection. However, they may lead to artifacts or incorrect results when rays pass through the back face of a triangle. However, 2-sided triangles allow for accurate intersection tests, as both the front and back faces are considered, but they may increase the computational complexity of the intersection tests.

It is crucial to consider the specific geometry requirements when choosing between 1-sided and 2-sided triangles. In Molflow, facets are always flagged as being 1-sided or 2-sided. Though with NVIDIA OptiX, we have to define for the whole geometry, whether face-culling (1-sided) should be applied or not. In OptiX, back-face culling is implemented as a ray property `OPTIX_RAY_FLAG_CULL_BACK_FACING_TRIANGLES`. The ClosestHit kernel does not differentiate further between the types. An initial prototype showed that treating each facet as 2-sided and to handle the appropriate effects (facet ignore) is more complex than treating each facet as 1-sided, we focus on the latter case. Here, 1-sided facets don't demand any particular treatment. To model a 2-sided facet, a clone of a facet can be created that has a flipped facet normal. We tested this method by putting a plane in the middle of a cylindrical tube, that is splitting the geometry in two. This approach didn't show any negative impact on the accuracy of the results compared to a CPU simulation and is thus used as the standard for the GPU kernel.

8.3.2.6 Facet types

In our initial GPU implementation, we only differ between two types of facets: solid and transparent. Transparent facets were briefly introduced in chapter 2.3.1. The opacity of a transparent is given by the opacity value $\tau \in [0, 1]$. A solid facet has a value of $\tau = 0$, while $\tau = 1$ denotes a perfectly transparent facet. Semi transparent facets with $\tau \in (0, 1)$ are special cases, where we have to use a randomly generated number $\text{R} \in [0, 1]$ to determine, whether it is treated as a solid or a transparent facet.

In OptiX, we design this partially with the Shader Binding Table (SBT), as introduced in chapter 8.2.2. We implement two kernels `closest_hit` and `closest_hit_trans`. With the SBT, we can create something similar to a function pointer. We map the `closest_hit_trans` kernel to all transparent facets ($\tau = 1$) and `closest_hit` to the others. When an intersection occurs, the SBT will call the appropriate kernel for the corresponding facet type. Semi-transparent facets are a special case, which we currently consider in `closest_hit`. This is because we have to decide during runtime whether the facet is using post-processing routines corresponding to a solid or a transparent facet. We considered developing a third type `closest_hit_semi_trans` to handle these cases, but did not investigate any further because most of our test cases either used solid or fully transparent facets.

8.3.2.7 Ray generation

Molflow's original Monte Carlo model (see chapter 2.2.3) has been designed with CPU architecture in mind. To leverage the strengths of the GPU and the RT cores, some modifications to the original model have to be made. Here, we revise the process in finding the starting parameters from chapter 2.2.3 by applying certain modifications related to deploying a triangle mesh.

When handling a new particle, the `Ray generation` program will initially generate the starting parameters according to Molflow's physical model. The minimal set of attributes for a ray tracing engine consists of the source location (`ray origin`) and the particle trajectory (`ray direction`), where Molflow also accounts for the particle velocity. Here, the velocity is only used for the post processing step. With the starting parameters the actual ray-tracing can be initiated.

Compared to Molflow's original implementation, the ray generation program for the GPU also handles different particle states. This is due to the nature of OptiX' ray tracing pipeline (see figure 8.3). In our implementation we handle the particle states: NEW_PARTICLE, ACTIVE_PARTICLE, TRANSPARENT_HIT and SELF_INTERSECTION. NEW_PARTICLE handles the default case, where a new particle is instantiated with starting parameters according to the model. ACTIVE_PARTICLE are particles that reside inside the system e.g. after a collision. In case of a previous collision, the particle remains inside the system and gets reflected. Here the set of attributes can simply be fetched from memory to initiate the particle for the ray-tracing routine. For OptiX, in such scenarios, the ray generation step can be potentially skipped via recursion (see figure 8.3). After a reflection has been processed, a recursive launch will directly launch the ray traversal. This is further elaborated in chapter 8.3.2.9. TRANSPARENT_HIT and SELF_INTERSECTION are special cases, that behave similarly as ACTIVE_PARTICLE in the ray generation step. Here, the previous state is simply initialized. The exact behaviour of these states will be elaborated later.

Ray origin: On a mesh, the source location has to be found in two steps (compare section 2.2.3 for more details). First, given the local influx rate of particles dN_f/dt for each facet, we can determine the probability that a particular facet f_i will be chosen as the starting location. If $\dot{n}_\Sigma = \sum_{i=1}^{N_f} \dot{n}_i$ is the total flux rate of the whole system, we get the probability that a facet f_i should be chosen for outgassing with $p_{Q_i} = \dot{n}_i/\dot{n}_\Sigma$, where $\dot{n}_i$ is the influx rate for facet f_i and $\dot{n}_\Sigma$ is the total influx rate for all facets. Given these probabilities, we create a CDF. Using this CDF of the artificial influx probabilities, we pick a random outgassing facet with a random number $\mathrm{X} \in [0, 1]$.

Selecting the exact outgassing location from a chosen facet, differs slightly for polygons and triangles. For a polygon this can be done by generating a random point inside the bounding rectangle and by accepting or refusing the candidate point with a point-in-polygon test. In certain cases, this could lead to multiple point generations until a ray origin could be determined. For a triangle, we select a random position with a different principle. A random point inside a triangle is then selected according to a formula described by Osada et al. (2002). Given a triangle with vertices $\mathbf{A}, \mathbf{B}, \mathbf{C} \in \mathbb{R}^3$ and two uniformly distributed random numbers $r_1, r_2 \in [0, 1]$, a point p can be sampled from a uniform distribution with the following equation:

$$\mathbf{p} = (1 - \sqrt{r_1}) \cdot \mathbf{A} + \sqrt{r_1} \cdot (1 - r_2) \cdot \mathbf{B} + \sqrt{r_1} \cdot r_2 \cdot \mathbf{C}\,. \tag{8.3}$$

$\sqrt{r_1}$ gives the percentage from vertex $\mathbf{A}$ to the opposing edge connecting vertices $\mathbf{B}, \mathbf{C}$ and r_2 denotes the percentage along that edge. The relation between the random numbers and the vertices is sketched in figure 8.8.

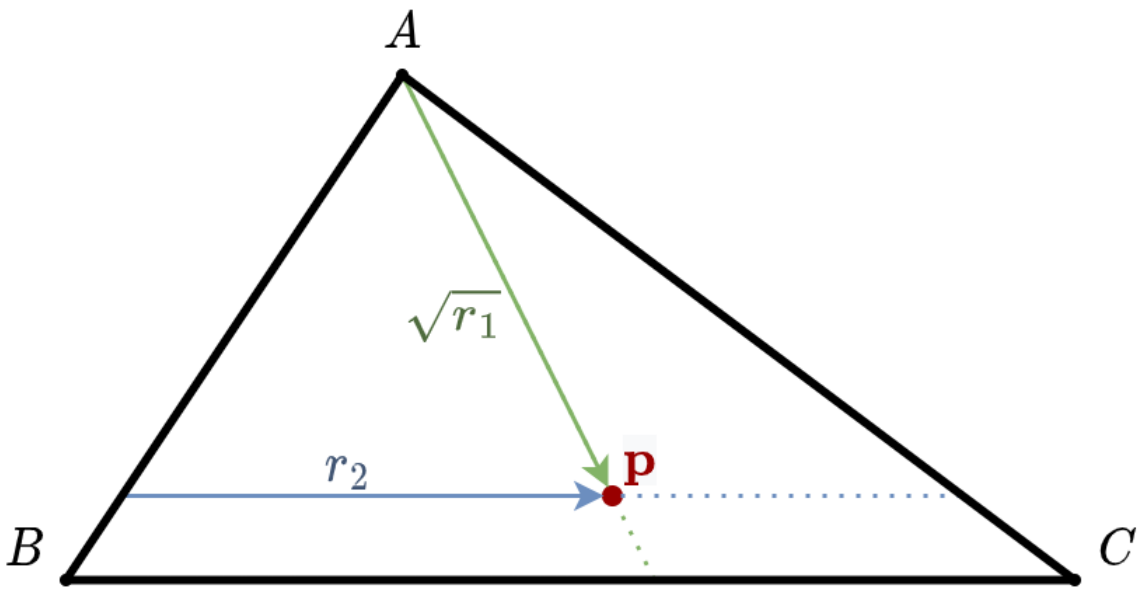


Figure 8.8: A point $\mathbf{p}$ can be sampled with two random numbers from a uniform distribution (cf. Osada et al., 2002). Here, $\sqrt{r_1}$ represents the distance in percent from vertex $\mathbf{A}$ to the opposing edge. r_2 represents the distance in percent from the edge $(\mathbf{A}, \mathbf{B})$ and the edge $(\mathbf{A}, \mathbf{C})$.

Ray direction: The calculation of the particle direction follows the same equations based on Knudsen's cosine law as elaborated in chapter 2.2.3. For completeness, we describe the only steps used to compute a random direction based on two random numbers. One can derive the azimuth and polar angles (ϕ respectively θ) with equations (2.13) and (2.14):

$$\begin{aligned} \phi &= \mathrm{X} 2\pi \quad , \mathrm{X} \in [0,1]\,, \\ \theta &= \sin^{-1}\sqrt{\mathrm{Y}} \quad , \mathrm{Y} \in [0,1]\,, \end{aligned}$$

where X and Y are uniformly distributed random numbers. For use in a 3-dimensional Cartesian system, one has to transform the derived spherical coordinates further into

Cartesian coordinates $(1, \theta, \phi) \to (x, y, z)$. First we derive the local Cartesian coordinates (u, v, n):

$$\begin{aligned} u &= \sin(\theta) \cdot \cos(\phi)\,, && (8.4) \\ v &= \sin(\theta) \cdot \sin(\phi)\,, && (8.5) \\ n &= \cos(\theta)\,, && (8.6) \end{aligned}$$

and translate them with the corresponding facet's orthonormal basis $\{\mathbf{U}, \mathbf{V}, \mathbf{N}\}$, where $\mathbf{N}$ is the facet normal, into global Cartesian coordinates (x, y, z) for the particle direction:

$$\begin{aligned} x &= u \cdot U_x + v \cdot V_x + n \cdot N_x\,, && (8.7) \\ y &= u \cdot U_y + v \cdot V_y + n \cdot N_y\,, && (8.8) \\ z &= u \cdot U_z + v \cdot V_z + n \cdot N_z\,. && (8.9) \end{aligned}$$

Particle Velocity: The calculation for the particle velocity works identically for the GPU kernel as for the original CPU-based code described in chapter 2.2.3. For simplicity, we use the average velocity for the Maxwell-Boltzmann distribution given by equation (2.21) for an initial study for both CPU and GPU simulations to validate the results.

Further, we implemented the generation of a random velocity from a CDF given by equation (2.24). For the GPU, on device memory we create two arrays `cdf_1` and `cdf_2`. Here, `cdf_1` contains all values v_i and `cdf_2` contains all values p_i. Further, we concatenate the values for all CDFs each corresponding to a specific temperature value. So for K CDFs, `cdf_1` and `cdf_2` have $N \cdot K$ values. To find the CDF values for the i-th bin and the k-th CDF, we simply restrict the search to the interval of indices $[k \cdot N, N + k \cdot N)$. Given the CDF k and a random value R, we use algorithm 8 to calculate a random velocity.

Ray launch: The RTX unit works with single-precision floating point values, which can have various negative effects on the ray-tracing results, most simply leading to occasional misclassifications: Given that the RTX intersection algorithm is guaranteed to be watertight in the sense that a ray can never go between two adjacent triangles which share the same edge, it will choose either one or the other for the hit location. If the ray passes very close to this edge, then the intersection may be attributed to the wrong triangle. For rendering problems, this is rarely noticeable when the color of a single pixel is slightly darker or brighter than expected. In Molflow this can lead to more severe problems, as the magnification of these errors can easily influence the whole system, leading to wrong results. We elaborate more on these problems and a potential solution in chapter 8.4.

Algorithm 8 Calculate random velocity from CDF

```
 1: procedure RANDOMVELO($k$, $N$, cdf_1, cdf_2, R)
 2:     $v \leftarrow$ cdf_1                                    ▷ Velocity values
 3:     $p \leftarrow$ cdf_2                                    ▷ Probability values
 4:     $l \leftarrow k \cdot N$
 5:     $h \leftarrow l + N$
 6:     while $l < h$ do                                        ▷ Binary search
 7:         $mid \leftarrow \lfloor 0.5 \cdot (l + h) \rfloor$
 8:         if $\textsc{r} \leq p_{mid}$ then
 9:             $h \leftarrow mid$
10:         else
11:             $l \leftarrow mid + 1$
12:         end if
13:     end while
14:     $j \leftarrow l - 1$
15:     $j \leftarrow \max(0, \min(j, N - 2))$                  ▷ Clamp $j$ within valid range
16:     $\Delta p \leftarrow p_{j+1} - p_j$
17:     $\epsilon \leftarrow \textsc{r} - p_j$
18:     return $v_j + (v_{j+1} - v_j)\frac{\epsilon}{\Delta p}$
19: end procedure
```

Further problems arise when we compute the exact hit location and take this value as the next ray origin to relaunch a ray from the corresponding surface: a secondary ray. The new location could be classified on either side of the polygon due to arithmetic error, as discussed in great detail by Wächter and Binder (2019). Potential workarounds differ with the use of back-face culling. A polygon's front is identified by one of its surface normals, that is the perpendicular vector corresponding to the surface. Without culling, both faces of a polygon serve for possible intersection points. With back-face culling, an intersection is only valid if the surface normal is facing the ray direction.

When we deploy back-face culling, there are few possibilities of what could happen in case the ray origin is found to be outside. Ideally, it will simply get redirected into the body of the mesh. Worst case, the hit location is on the edge between two non-coplanar triangles and the ray direction is quasi-parallel to the neighboring triangle plane. In this case the ray will not find a collision point with the neighboring triangle: a miss.

In addition to back-face culling, we can reduce the amount of misses of arithmetical origin by deploying an adaptive offset along the facet normal. We chose the strategy presented by Wächter and Binder (2019) as it leads to minimal computational overhead and minimal impact on the accuracy compared to other options. The authors empirically analysed the error of floating point calculations in ray-triangle intersections. Their analysis concluded that both the distance between the intersection point and the space origin $(0, 0, 0)$ as well as the size of the surface have an impact on the numerical point error. Given their results, they give a robust offset based on the maximum distance between the actual intersection point and the maximal numerical error. An offset along the orthogonal normal is chosen as it will feature the smallest possible offset, having a limited effect on the results. Given a point in 3d coordinates $\mathbf{p}$ and the facet normal $\mathbf{N}$ in 3d space, an offset $p_{o,\omega}$ can be calculated with:

$$\delta = \iota \cdot N_\omega \,, \tag{8.10}$$

$$p_{i,\omega} = p_\omega +_{INT} \begin{cases} +\delta & \text{if } p_\omega \geq 0 \\ -\delta & \text{if } p_\omega < 0 \end{cases} , \tag{8.11}$$

$$p_{o,\omega} = \begin{cases} p_{i,\omega} & \text{if } p_\omega \geq \epsilon \\ p_\omega + \zeta \cdot N & \text{if } p_\omega < \epsilon \end{cases} . \tag{8.12}$$

Here, index $\omega \in x, y, z$ represents the component for the corresponding coordinate axis. The authors give empirically found values for $\iota = 256.0$, $\zeta = 1/65536$, and $\epsilon = 1/32$. In equation (8.11) they make use of integer arithmetics (noted with $+_{INT}$ as integer arithmetic on real numbers) so that the offset becomes scale-invariant "preventing self-intersections at distances of different magnitudes". ϵ is used as threshold to determine if the point $\mathbf{p}$ is very close to the origin $(0,0,0)$ in any of its components x, y, or z. ι is used to create a small offset 2^{-16} for the point $\mathbf{p}$ when it is very close to the origin. ζ is a scaling factor used to convert the normal vector components to a larger scale, making them suitable for conversion to integer values. Algorithm 1 shows the implementation proposed by the authors.

Listing 1 Ray offset calculation as proposed by Wächter and Binder (2019).

```
constexpr __device__ float origin()      { return 1.0f / 32.0f; }
constexpr __device__ float float_scale() { return 1.0f / 65536.0f; }
constexpr __device__ float int_scale()   { return 256.0f; }

// Normal points outward for rays exiting the surface, else is flipped.
static __forceinline__ __device__ float3 offset_ray(const float3 p, const
    float3 n){
    int3 of_i(make_int3(int_scale() * n.x, int_scale() * n.y, int_scale() * n.
        z));
    float3 p_i(make_float3(
            int_as_float(float_as_int(p.x)+((p.x < 0) ? -of_i.x : of_i.x)),
            int_as_float(float_as_int(p.y)+((p.y < 0) ? -of_i.y : of_i.y)),
            int_as_float(float_as_int(p.z)+((p.z < 0) ? -of_i.z : of_i.z))));
    return float3(make_float3(
            fabsf(p.x) < origin() ? p.x+float_scale()*n.x : p_i.x,
            fabsf(p.y) < origin() ? p.y+float_scale()*n.y : p_i.y,
            fabsf(p.z) < origin() ? p.z+float_scale()*n.z : p_i.z));
}
```

In an experiment with the introduced cylindrical tube approximated with 1000 vertices per end cap we found that back-face culling itself resulted in a miss-hit ratio of $1.35 \cdot 10^{-7}$ and the adaptive offset in a ratio of $1.44 \cdot 10^{-6}$. Deploying both solutions reduced the rate to only $5.32 \cdot 10^{-11}$ for this geometry. With neither back-face culling nor an offset almost half of the particles leak. This is because already during initialization they could numerically spawn on the outside, leading to the problem depicted in figure 8.4. This improvement was visible in experiments for variations of the cylindrical tube. Therefore, this was used as the default strategy for further simulations. We found later, that the remaining leaks are mainly related to a problem, that is caused by the applied offset itself. This problem is further investigated in chapter 8.4.

Possible consequences for the given geometry in case of a miss can be neglected. Theoretically, if there is a surrounding shell, the ray would hit that surface instead, leading to a series of unexpected hits. Depending on the complexity of a geometry, this can lead to uninterpretable results.

The ray generation procedure is sketched in algorithm 9. A particle is either initialized with the described model (*new_particle*) or with its state from a previous launch[6]. After the offset has been applied, the particle is launched with a call to the

[6]Here, the other states are equal for the for ray generation and only differ in their post processing step. They are listed individually for clarification.

Algorithm 9 Ray generation algorithm

```
1: function RAYGENERATION(Particle p)
2:     ($p.terminate$ == true) → return
3:     switch $p.state$ do
4:         case $new\_particle$
5:             INIT_STATE()
6:         case $active\_particle$
7:             LOAD_STATE()
8:         case $transparent\_hit$
9:             LOAD_STATE()
10:        case $self\_intersection$
11:            LOAD_STATE()
12:    APPLY_OFFSET(p.position, p.previous_facet.n)
       ▷ Launch ray with OptiX API; calls RTX pipeline to find intersection
13:    OPTIXTRACE()
14: end function
```

OptiX API and to trace the ray and to identify a potential intersection. Initially (see line 2) a check is conducted to see whether a thread can terminate early due to a fulfilled end condition, which is set on the CPU.

8.3.2.8 Intersection test

RT cores accelerate bounding volume hierarchy (BVH) traversal as well as ray-triangle intersection via hardware. For our study we investigate both triangle and polygon meshes (see chapter 8.3.2.3). In the case of triangle meshes, the initial polygon mesh in Molflow has to be triangulated and transfered to the device via the OptiX API. For triangles we use the built-in intersection routine. Here, both BVH traversal and intersection tests are hardware-accelerated.

On the other hand, polygon meshes need further consideration. To utilise hardware-accelerated BVH traversal for polygons, a set of axis-aligned bounding boxes have to be provided for each facet. Further, a custom implementation of the two-step ray-polygon test routine has to be provided. We ported the original routine used in Molflow (see chapter 2.3.5). First, a ray-rectangle intersection is used, where we solve a system of linear equations using Cramer's rule (see Shirley and Marschner, 2009). This is followed by a point-in-polygon test in 2D space, where the ray-casting algorithm provided by W. R. Franklin (2018) provided the best single-precision results on the GPU in both performance and reliability in terms of the false negative rate in our experiments.

8.3.2.9 Trace-processing

Following the calculation of a potential hit location, the events are accumulated in hit counters for statistical purposes. The gathered statistics correspond not only to the plain amount of collisions or MC Hits N_{hit}, but also to the sum of orthogonal momentum changes of particles $\Sigma I_\perp$ and the sum of reciprocals of the orthogonal speed components of particles $\Sigma v_\perp^{-1}$. From these three counters the most important values for vacuum components can be derived, e.g. the pressure or density as explained in chapter 2.2.3.

The counters are increased according to the corresponding collision event related to the facet type. Besides the per-facet hit counters, the users can also enable more fine-grained counters in the form of textures $(N \times M)$ or profiles $(1 \times N)$, which divide a facet in two directions or one direction respectively. Here $N, M \in \mathbb{N}_+$ describe the spatial resolution. If used, an extra step to calculate the index of the texture element or profile bin has to be deployed. Texture and profile coordinates are given in local coordinates $(u, v) \in [0, 1]^2$ in 2D space, which simplifies the indexing of the underlying data structure. The coordinates are already provided by the polygon intersection test as they are part of the calculation. For a texture mapped onto a triangle we provide local texture coordinates for each vertex $\mathbf{A}, \mathbf{B}, \mathbf{C} \in [0, 1]^2$ and use the barycentric coordinates returned by the intersection test to interpolate the exact local hit location using barycentric interpolation (8.1).

In our simplified model, we differentiate between two types of collision events based on a target facet's properties. If a particle gets absorbed, new particles will be generated in the next ray generation step. In case of a collision with a solid surface, the particle is reflected. The hit location can be reutilised as source location, thus only a new ray direction needs to be generated according to the cosine law (2.11). Instead of postponing this to the next ray-generation step, a new ray can be recursively launched from the shading routine. In rendering applications this is commonly done for so-called secondary rays, simulating light reflections on surfaces. These secondary rays are launched recursively up to a pre-specified recursion depth. Skipping the ray-generation step is cheaper, but comes with some restrictions. In algorithm 10 we show a sketch of the trace processing algorithm. The corresponding RECORD_ routines account for both global and local statistics.

As the particles in Molflow tend to stay inside a given geometry for several hundreds or thousands of collisions on average, the full recursion would easily reach the maximum stack size of a thread. Given that some particles could terminate, while others could be recursively retraced, some idle time is expected. A well chosen value for a maximum recursion depth will likely have a positive impact on overall performance either way, leading to a hybrid approach, where residual particles, that are those that have not been absorbed after a ray-tracing step, are reinitialized inside the ray generation program after the maximum recursion depth has been reached.

Algorithm 10 Trace processing algorithm

```
1: function TRACEPROCESSING(Particle p)
2:     if hit same polygon again? then
3:         $state \leftarrow active$                              ▷ Try again with new offset
4:         return
5:     end if
6:     MOVE_PARTICLE()                                           ▷ $\mathbf{o}_{ray} + \mathbf{r_{dir}}t$
7:     if particle sticks? then
8:         RECORD_ABSORPTION()                                   ▷ record statistics
9:         $state \leftarrow new\_particle$
10:        return
11:    else if particle reflects? then
12:                              ▷ record statistics and update particle direction
13:        RECORD_REFLECTION()
14:        if do recursion? then
15:            APPLY_OFFSET(p.position)
16:            OPTIXTRACE()
17:                                      ▷ Recursive launch ray with OptiX API
18:        else
19:            $recursion\_depth \leftarrow 0$
20:        end if
21:    end if
22: end function
```

We benchmark this technique in the next section.

8.3.2.10 Random numbers and recursion

A crucial part of a good Monte Carlo simulator is the underlying PRNG algorithm (see chapter 2.2.5) in both credibility and performance. For the best results, we utilize the cuRAND library (see NVIDIA, 2019) from the CUDA toolkit and its default PRNG Xorwow (see G. Marsaglia, 2003). To deploy the random numbers there are two fundamental approaches. Most straightforward, each GPU thread has its own random state `curandState_t` of 48 bytes for the Xorwow PRNG to generate random numbers on an ad hoc basis. A different approach is to batch generate multiple random numbers in advance every few cycles in a separate CUDA kernel.

The first approach is memory friendly, as we only have to save a random state instead of $N_{max} \times T_C$ random numbers per thread, where N_{max} is the maximal amount of random numbers needed for one ray tracing cycle and $T_C \in [1, \infty]$ describes the number of cycles to generate numbers for in advance. All random numbers are generated in a static fashion for all threads in a separate CUDA kernel on device memory. An additional counter keeps track of the index for the next random number. Depending on how an interaction with the boundary affects the particle, not all numbers have to be accessed. With batch-wise generation an additional memory amount of $(N_{max} - N_{min})$ random numbers per thread has to be accounted for, which can remain unused. Here, N_{min} is the minimal amount of random numbers. For example, in case a particle remains inside the system the particle origin does not have to be recalculated with two random numbers. When using recursion, more random numbers have to be generated in advance. For a full cycle Molflow's algorithm utilizes up to 8 random numbers: 3 to find the particle origin, 2 to find the ray direction and 1 to account for variable sticking coefficients. If we use recursion, the ray direction needs to be calculated in the trace processing step adding another 2 to find the ray direction. Skipping ray generation only 2 random numbers are needed to compute a new ray direction, leading to $N_{max} = 8 + \textsc{recursion\_depth} \times 2$ random numbers per cycle and ultimately to $T_C \times N_{max}$ buffered random numbers.

In section 8.3.3.3 we analyze how the usage of both approaches for random number generation affects the performance with varying recursion depth limits. We would like to note, that this experiment was conducted as part of Bähr et al. (2022). The experiment did not consider additional random numbers required for the particle velocity (simply the average velocity has been used) and semi-transparent facets, which are evaluated in the trace-processing step, potentially leading to $N_{max} = 10 + \textsc{recursion\_depth} \times 4$ random numbers per cycle.

8.3.2.11 Device memory

Molflow geometries tend to be memory space efficient and are ultimately shared between all threads. We can divide the remaining memory among all particles, which we want to launch in one go. For every thread the memory demand is usually not fixed, and depends on the deployed design solutions. Accounting for about 64 B for the particle state (here, B=Bytes) and an additional 48 B for a RNG state or $T_C \times 24\,\mathrm{B} + 4\,\mathrm{B}$ for using batch generation with 6 single-precision random numbers per cycle and an additional index for the next random number, the remaining memory can be used for the hit buffers. To prevent race conditions or excessive access synchronization every thread ideally has its own hit buffer, per facet that amounts to $6 \times 4\,\mathrm{B}$ for 6 counting variables (see chapter 2.2.3). In addition, when utilizing textures or profiles for data collection on a more fine grained scale, we have to add $3 \times 4\,\mathrm{B}$ per texture element or profile bin. Obviously, having individual counters per thread increases performance, since we don't have to utilize any sort of synchronization, e.g. in the form of atomics.

For example, considering a simulation on an NVIDIA RTX 2060 as the reference

GPU, we found deploying around 128×16384 threads (see section 8.3.3.1) to saturate the GPU nicely. Now, given a simple pipe geometry with a total of only 102 total facets[7] as an example. We would need to allocate around $128 \times 16384 \times (64+36)\mathrm{B} = 210\,\mathrm{MB}$ for the states of the rays and additionally around $5134\,\mathrm{MB}$[8] if every thread would have its own hit counters, thus easily reaching the full 6 GB VRAM capacity, without deploying any additional fine-grained counters. This is obviously not a feasible strategy, as the memory limit would have already been reached with the other memory requirements. Therefore, we consider to utilize one shared facet buffer by default, which is modifiable with atomic operations. In section 8.3.3.2 we analyze the option to use multiple buffers among a group of threads.

8.3.3 Performance and precision study

To account for all types of Molflow users, we decided to provide benchmarks for two sets of hardware. The first set accounts for the average Molflow user, where we consider consumer-grade hardware as found in laptops or office computers, on which simulations in early phases are usually run. We utilize an entry-level Turing GPU with RTX technology of the first generation. With a stronger emphasis on complex and time-demanding simulations, the second set focuses on high-end hardware with a second generation RTX GPU:

- Set 1: {CPU Intel i7-8557U, GPU NVIDIA RTX 2060} ,
- Set 2: {CPU AMD Epyc 7302P, GPU NVIDIA RTX 3090} .

The software was compiled with GCC 10.2 and level 3 optimizations on all machines, running an Ubuntu 20.04 based system. The GPU kernel was compiled with OptiX 7.1 and CUDA 10.1 . As input geometry, all experiments utilize a cylindrical tube, as it was previously introduced in section 4.3. Using the cylindrical tube with varying parameters, the development of the GPU kernel can be validated and benchmarked without being exposed to too many application specifics in a first step. We would like to note, that the experiment was conducted as part of the work by Bähr et al. (2022). The implementation was only able to handle simple geometries as techniques required to simulate more complex test cases, were developed later. They are introduced in chapter 8.4.

For simplicity, we will refer to these pipes with $\mathcal{G}_{lr,n}$, where index lr denotes the length / radius ratio and index n denotes the number of side facets used to approximate the circular shape. For most tests a pipe with a length/radius ratio

[7]Note, that for a triangle mesh we map triangles onto their corresponding parent polygons to prevent allocating extra hit buffers per triangle.

[8]For 6 counting variables à 4 Byte, this is 24B per facet. Now 128×16384 threads have 24B for 102 facets.

of $L/R = 100$ and either 100 or 1000 side facets are used: $\mathcal{G}_{100,100}$ and $\mathcal{G}_{100,1000}$, respectively. This configuration has an analytic solution, which can be used as a benchmark. The inlet (desorption facet) defines a steady influx of particles. Furthermore, both end facets serve as perfect absorbers, removing all particles from the system. For the side facets, we define them as perfectly diffusive, reflecting all particles. Algorithmically, we utilize both the built-in intersection routine provided by the OptiX API and a custom polygon intersection routine. Using the former, one can see hardware-acceleration effects on both BVH traversal and intersection testing, whereas for the custom routine only BVH traversal can benefit from these effects. For a fair comparison with the CPU algorithm of Molflow+, we utilize an updated version of the code, with major improvements to the BVH structure and intersection routine.

8.3.3.1 Amount of threads

First we run a simple experiment to find how different amounts of threads launched simultaneously influence the performance. We use power-of-two multiples of 128 threads, which represents the amount of CUDA cores per streaming multiprocessor for the RTX 3090, scaling up to 2^{25} threads in total.

In figure 8.9 we can see, that the performance on the overall faster RTX 3090 is unsteady for the simulation of the $\mathcal{G}_{100,100}$ geometry. It is an effect of the atomic operations for the hit counters, which we investigate in the next experiment. Further we can see that the performance increase stagnates between $128 \times 8192 = 1,048,576$ and $128 \times 16384 = 2,097,152$ simultaneous threads for the NVIDIA RTX 3090 and the RTX 2060 respectively, for both geometries. This seems to be the sweet spot, when the GPU is saturated enough. We can choose these values for memory critical simulations.

8.3.3.2 Extra buffers

As described in section 8.3.2.11, atomic operations can be used to effectively counter race conditions. Considering that it is possible that certain facets are frequently hit, using a single hit buffer for each facet can save memory, but negatively impact the overall performance. We investigate the benefits of utilizing multiple buffers per facet, where the buffers are split equally among all threads in a warp.

For the $\mathcal{G}_{100,100}$ geometry, figure 8.10 shows that increased performance can have atomic operations as a bottleneck as the probability of simultaneous access increases. The effect can be reduced by utilizing multiple counters instead of only a single one, where 4 counters are sufficient in this case. On average, a triangle describing a side facet has a hit chance of around 0.45% per step, decreasing the chance to 0.11% for one of four counters. As the effect was hardly visible with a slower GPU (RTX 2060)

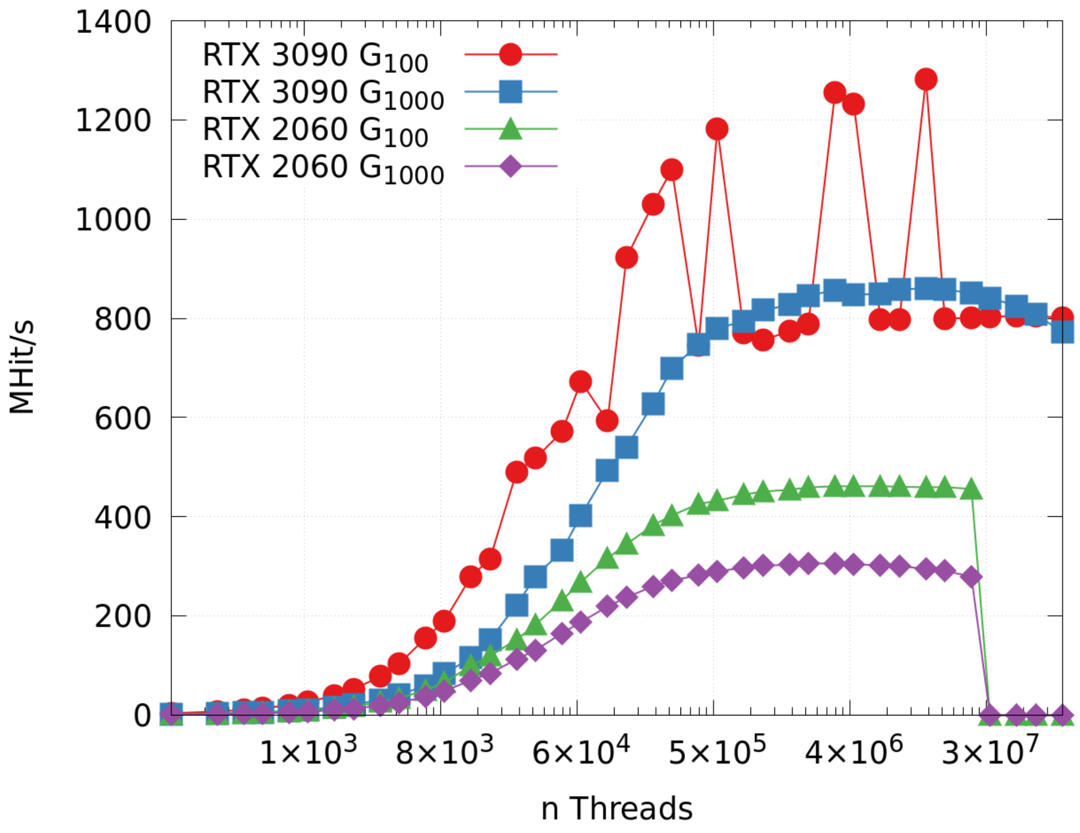


Figure 8.9: Performance measured in Hits per second (1 MHit = 10^6 Hit) in relation to simultaneously launched threads, where the x-axis is $\log_2$-scaled. Memory limits are reached for higher thread numbers on the RTX 2060, resulting in 0 MHit/s.

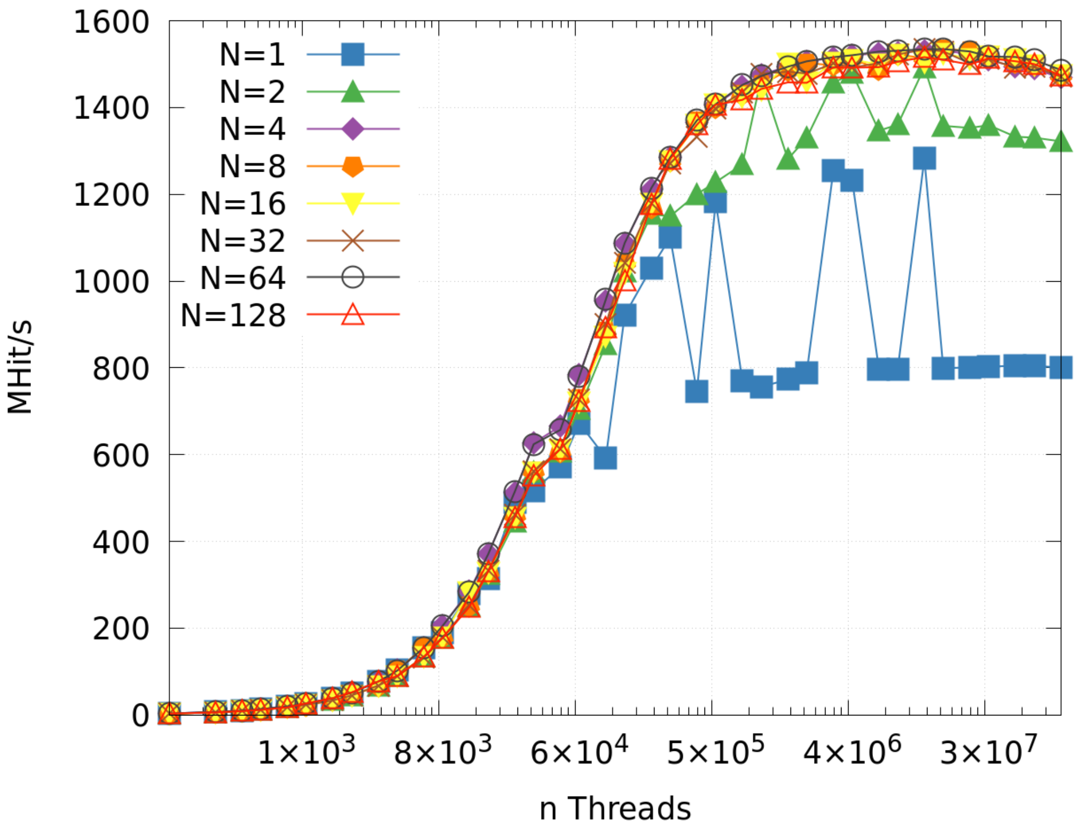


Figure 8.10: Effect of atomic operations in combination with a variable amount of facet buffers N on the performance, given more simultaneously launched threads, for the $\mathcal{G}_{100,100}$ geometry simulated on an NVIDIA RTX 3090.

or a geometry with more facets ($\mathcal{G}_{100,1000}$), because of the lower hit frequency per facet, it is not necessary for most geometries to utilize multiple counters as memory is likely more of a concern.

8.3.3.3 Recursion and RNG

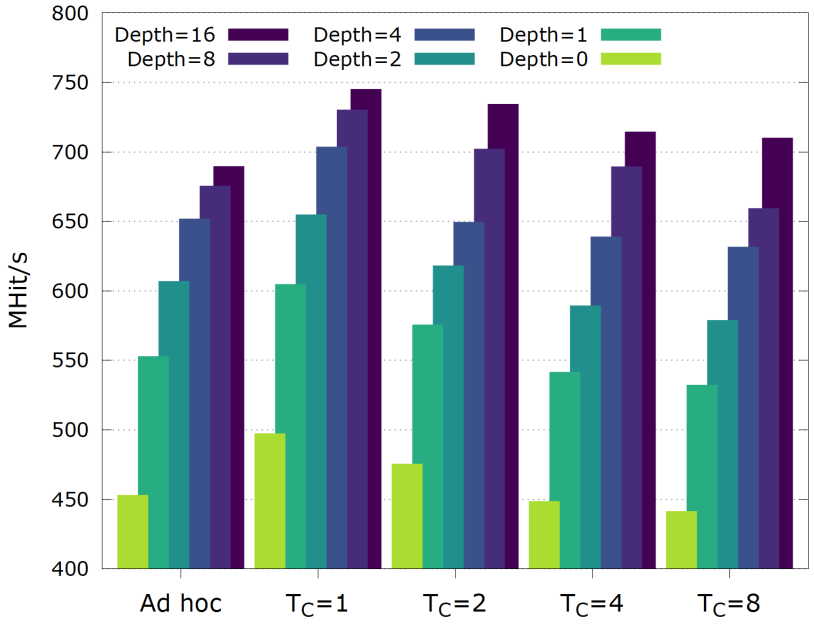
To evaluate the effects of the different approaches for random number generation – ad hoc and batched – and possible benefits coming from utilizing a recursive kernel for the launch of secondary rays in the case of reflections, we compared the results for different geometries ($\mathcal{G}_{100,100}$ and $\mathcal{G}_{100,1000}$) on both GPUs. The simulations are run with the corresponding thread numbers, which were found in the previous experiment: $128 \times 8192 = 1,048,576$ and $128 \times 16384 = 2,097,152$ for the RTX 3090 respectively the RTX 2060. For the geometry $\mathcal{G}_{100,100}$ we deploy 16 hit buffers for the simulations on the RTX 3090.

In figures 8.11 and 8.12 we see that random number generation in batches does yield better performance in all cases in conjunction with our ray tracing kernel. Surprisingly, generating more random numbers in advance does not speed up, but instead slightly slows down the simulations. We found that this is mostly cache related, where generation for one cycle already leverages the positive effects and multiple cycles slightly suffer from less hits in both L1 and L2 cache. For the RTX 3090 (figures 8.12a and 8.12a), the effect is less pronounced on a relative scale. The ad hoc generation of random numbers was inferior in all cases compared to the single cycle batch generation ($T_C = 1$). On the RTX 2060 (figures 8.11a and 8.11a) the relative difference between the ad hoc generation and the $T_C = 1$ batch generation is relatively close, which could be a reasonable choice when memory would be a problem for other geometries.

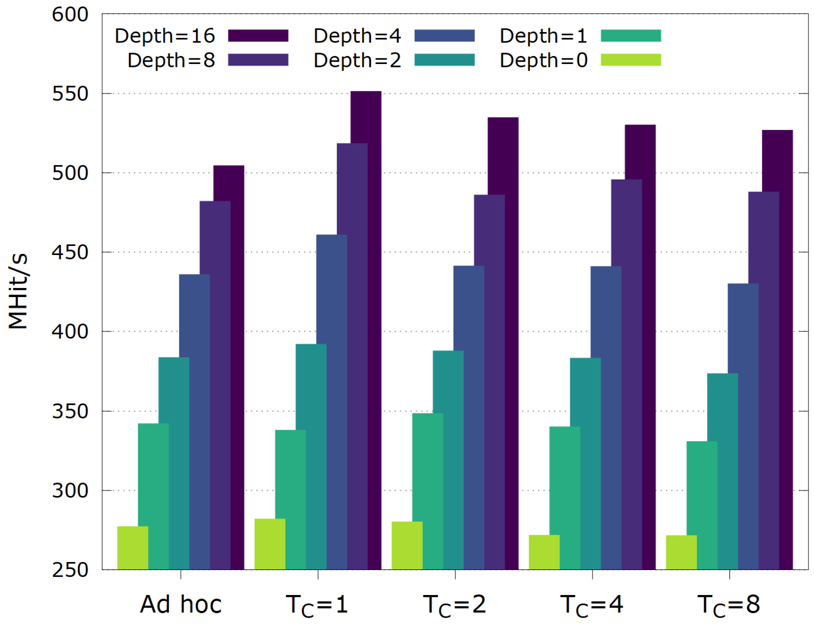
For vacuum geometries, particles usually reside inside the system for a large amount of collision events. This property makes them suitable for any level of recursion that can be deployed within the applicable memory constraints. For example, this is the case with the given test geometry, where on average each particle yields around $\sim 100\times$ events until it exits from the system. By contrast, a cylindrical tube with a ratio of $L/R = 1$ instead of $L/R = 100$ does not benefit a lot from recursion as particles yield for around $\sim 2\times$ events on average.

8.3.3.4 Performance

We analyze the raw performance of the ray tracing engine in two experiments. First, we run simulations on the specified geometry without any modifications to highlight the impact of the actual ray tracing on the performance. Next, we include textures on all facets to put a heavier load on the trace-processing kernel, which is closer to the needs of real-life simulations. In Molflow, statistical counters such as profiles or

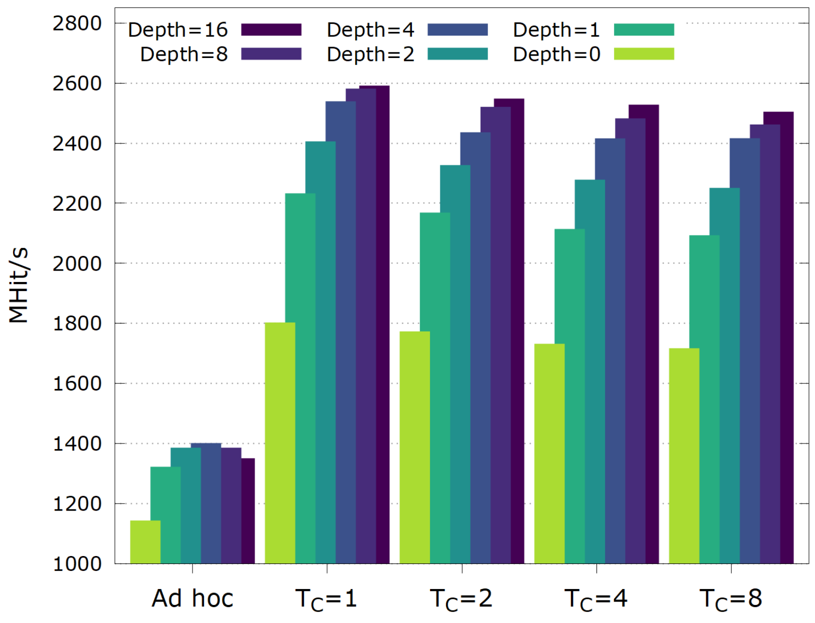


(a) RTX 2060: $\mathcal{G}_{100,100}$

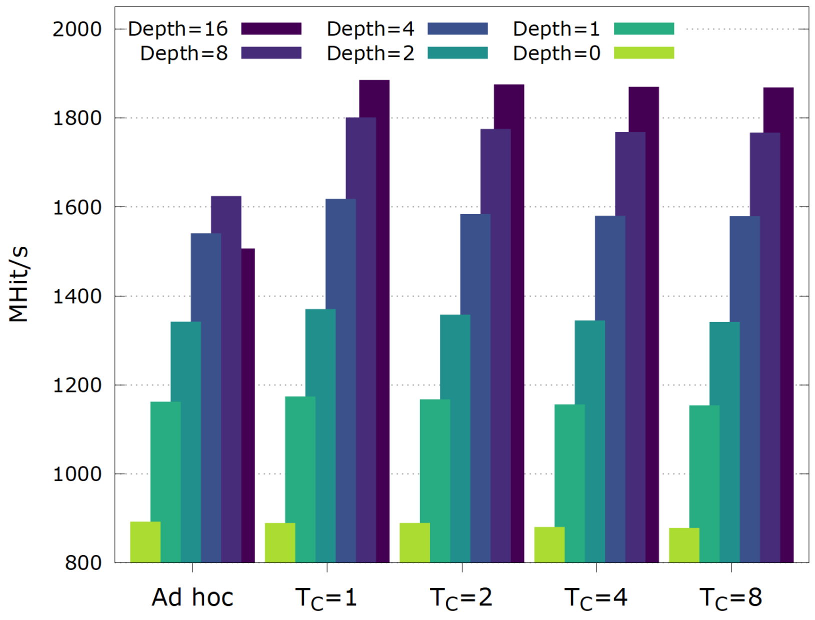


(b) RTX 2060: $\mathcal{G}_{100,1000}$

Figure 8.11: Performance for the Molflow GPU algorithm for a pipe with different configurations on NVIDIA RTX 2060. Results are generated for ad hoc and batched random number generation with varying cycles T_C and different levels of recursive depth.



(a) RTX 3090: $\mathcal{G}_{100,100}$



(b) RTX 3090: $\mathcal{G}_{100,1000}$

Figure 8.12: Performance for the Molflow GPU algorithm for a pipe with different configurations on NVIDIA RTX 3090. Results are generated for ad hoc and batched random number generation with varying cycles T_C and different levels of recursive depth.

Table 8.2: Performance measured in MRay/s (10^6) for the geometry with different approximations ($\mathcal{G}_{100,100}$ and $\mathcal{G}_{100,1000}$) and with (*w/*) or without (*w/o*) textures. The GPUs (RTX 2060 and RTX 3090) run on either a polygon mesh (*Poly*) or a triangle mesh (*Tri*). The CPUs (Intel i7-8775U and AMD Epyc 7302P) run on the original FP64 algorithm.

	i7-8775U	2060+Poly	2060+Tri	Epyc 7302P	3090+Poly	3090+Tri
$\mathcal{G}_{100,100}$ w/o tex	6.34	298.62	747.10	34.10	1308.18	2646.48
$\mathcal{G}_{100,100}$ w/ tex	5.74	274.10	575.10	32.27	1168.75	2003.73
$\mathcal{G}_{100,1000}$ w/o tex	4.56	172.48	538.55	10.07	602.03	1819.27
$\mathcal{G}_{100,1000}$ w/ tex	4.02	152.82	374.87	9.97	548.91	1347.90

textures are one of the standard techniques applied. For the simulations we utilize the same parameters as before. We generate random numbers with the single cycle batch generation method ($T_C = 1$) and a recursive depth limit of 16, which proved to be good parameters as previously shown in figures 8.11 and 8.12.

We can see in table 8.2 that we get good speed-ups for all test cases. The triangle-based algorithm is in all cases the most performant. The speed-up is largest for the consumer-grade hardware (set 1), for the $\mathcal{G}_{100,1000}$ geometry and no textures, showing that geometries with a focus on ray-tracing profit more from GPU utilization. Using textures has a bigger negative impact for the triangle-based GPU algorithm compared to the polygon-based algorithms on both GPU and CPU. This is likely related to the extra step in calculating the texture coordinates in 2D space. The RT algorithm for polygons on the CPU calculates the exact location as part of the intersection test. For the GPU algorithm the barycentric coordinates returned by the intersection routine are translated with texture coordinates of the corresponding vertices in an extra step (see section 8.3.2.9).

8.3.3.5 Precision

With some fundamental changes to the ray tracing algorithm, we also have to consider the possible impact on the accuracy of the simulation. With prior testing, we had analyzed the individual custom kernels (ray generation and trace processing) with a set of isolated simulations to identify a single point of error. We had concluded that only the intersection test, crucial for the calculation of the hit location, could possibly have a big impact. This is likely due to its 32 bit floating point limitation on the dedicated ray tracing units, which is a common problem for ray tracers (see Wächter and Binder, 2019).

Thus, for further conclusions, we included a modified CPU algorithm into our test set, which uses 32 bit precision for the geometry and ray description to correspond to the RTX hardware limitations. As the CPU algorithm has been found as unstable when run completely with 32 bit floating precision, leading to largely uninterpretable results, the remaining parts were kept with 64 bit precision.[9] We evaluated the transmission probability W after 10^9 desorptions[10] for 50 runs[11] for the CPU and GPU algorithm and compared them to an analytical solution for the transmission probability obtained from Gómez-Goñi and Lobo (2003) for a real cylindrical tube. The transmission probability is the ratio of particles that got absorbed on one end of the facet to the total amount of desorbed particles.

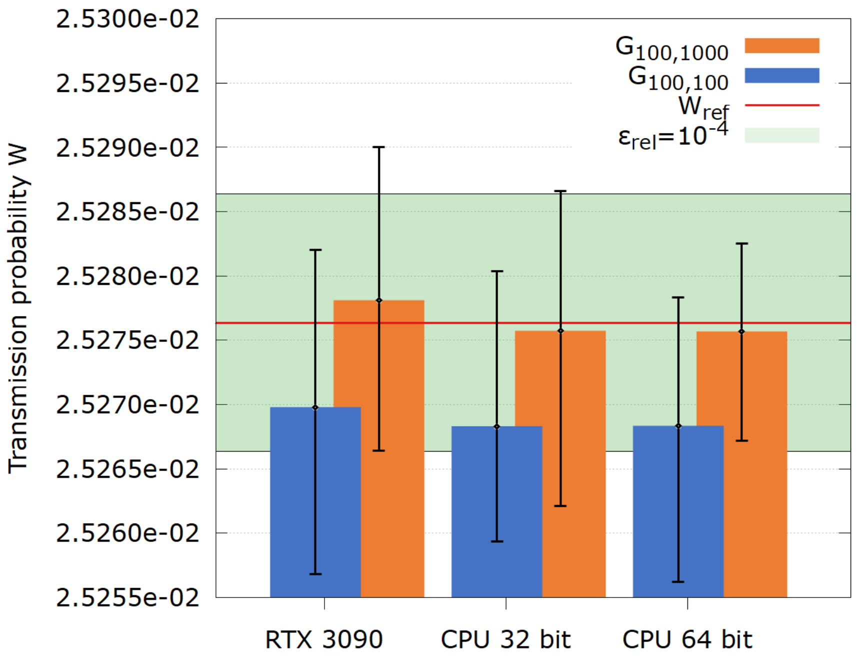


Figure 8.13: Transmission probability for a L/R=100 cylindrical tube for the GPU and CPU algorithm (with 32 bit and 64 bit geometry) in relation to an analytical solution $W_{ref} = 0.0252763636$ (see Gómez-Goñi and Lobo, 2003). Error bars denote the maximal and minimal value of the set, considering 50 simulations per set. The height of the blue and orange bars denotes the corresponding average result. The green bar is the relative tolerance region $\varepsilon_{abs} = 10^{-4}$ surrounding the analytical value.

As can be seen in figure 8.13, we find that the converged results are sufficiently

[9] The CPU algorithm has not been optimized for 32 bit computation, hence, no visible performance difference could be measured.

[10] Note, that one has to account for all particles to actually leave the system, as residual particles – especially in a large number like on the GPU – still contribute to the overall results.

[11] We consider the mean, min and max value of the set. The mean equals the probability after 5×10^{10} desorptions.

close on all architectures. Considering the mean values, the results of all methods converge towards the analytical solution staying at least within the $\varepsilon_{rel} = 10^{-4}$ error margin. Increasing the level of approximation, $\mathcal{G}_{100,1000}$, the results converge being close to the analytical solution W_{ref}. Calculations with 32 bit floating point precision does not seem to have a major impact on the precision, where the span between the minimum and maximum for each set are also comparable between the different architectures. Hence, if there is a demand for higher precision a better approximation of the geometry has to be considered first.

Table 8.3: Transmission probabilities $W_{\{GPU,CPU\}}$ for cylindrical tubes $\mathcal{G}_{lr,100}$ approximated with 100 side facets and varying L/R ratio, as calculated for a simulation with the GPU kernel respectively Molflow's CPU algorithm and 10^9 desorbed particles. The corresponding Δ values denote the absolute difference from the reference value W_{ref} obtained from Gómez-Goñi and Lobo (2003). The speedup for each set is given as $S = T_{CPU}/T_{GPU}$, where T denotes the corresponding execution time to simulate 10^9 particles. Set 1 represents low-budget hardware, and set 2 represents high-end hardware.

L/R	W_{GPU}	Δ_{GPU}	W_{CPU}	Δ_{CPU}	W_{ref}	S_{Set1}	S_{Set2}
1	0.67192400	5.99E-05	0.67201900	3.51E-05	0.67198390	63.20	31.09
2	0.51416070	6.98E-05	0.51406400	1.66E-04	0.51423050	66.04	30.75
5	0.31044060	1.88E-04	0.31041900	1.67E-04	0.31025230	68.32	30.47
8	0.22521630	4.68E-05	0.22526600	2.90E-06	0.22526310	126.56	36.68
10	0.19090100	4.14E-05	0.19087900	6.34E-05	0.19094240	121.59	40.08
20	0.10928520	3.55E-05	0.10932500	4.30E-06	0.10932070	114.16	49.34
50	0.04845371	2.27E-05	0.04847840	2.00E-06	0.04847640	76.77	49.69
80	0.03123623	1.11E-05	0.03123400	1.33E-05	0.03124730	174.41	49.54
100	0.02526948	6.92E-06	0.02527330	3.10E-06	0.02527640	164.04	49.67
200	0.01294084	5.40E-05	0.01293520	5.96E-05	0.01299480	118.35	48.78
500	0.00524651	1.65E-05	0.00524247	2.05E-05	0.00526300	133.14	49.65

Further, we show in table 8.3, that the behavior is similar for other geometries. We compare the results for simulations on cylindrical tubes with various length-radius ratios. The results differ for neither configuration by a significant amount besides the expected MC fluctuations. Here, we compared 64 bit CPU calculations against the GPU results.

8.3.3.6 Conclusion

We were able to show major speedups of for the developed GPU kernel, while proving minimal accuracy differences compared to the original CPU implementation for basic geometries such as cylindrical tubes. Our design achieved speedups of $63\times - 175\times$ on budget hardware and $30\times - 50\times$ on high-end hardware. While mitigating negative effects resulting from numerical errors and proving to have a low impact on overall performance, the adaptive offset showed not to be reliable as a stand-alone solution for more complex geometries. For cylindrical tubes it was able to reduce the amount of numerical leaks significantly. While this solved some effects caused by displaced ray origins, the offset itself can displace rays outside of other facets potentially leading to large statistical errors that can make simulations completely uninterpretable. In the next chapter, we provide a potential solution that relies on the adaptive offset, but attempts to fix the problems arising from that.

8.4 Neighbour Aware Offset (NAO)

In chapter 8.3.2.7 we advised utilising a strategy, that applies an offset along the facet normal to counter problems related to self-intersections and single-precision arithmetic. When the origin of a ray is placed behind the actual wall, it can lead to misclassifications. This can often occur due to numerical errors arising from single-precision calculations. In this case the ray might intersect with the same facet again, which either needs to be filtered out or leads to missed interactions with other facets. In OptiX, the performant CLOSEST HIT kernel does now allow to filter intersections for matching facet IDs, the default strategy deployed in Molflow's CPU engine. The ANY HIT kernel allows to compute results for every intersected face, which is computationally less efficient. There may still exist certain scenarios, where a rejection based on a matching ID can still occur in errors, e.g. when an intersection is located close to a shared edge of a two planar facets. This is elaborated in great detail by Wächter and Binder (2019).

Other solutions can be deployed to solve such problems. A ray identifies the closest intersection by requiring that the distance t hit location lies in the interval $t \in [t_{min}, t_{max}]$. t_{max} is the current maximal distance, typically initialized with an infinite value and updated by tracking the closest hit distance in the current search. By setting a minimum threshold $t_{min} = \varepsilon > 0$, the rays are prevented from intersecting with the surface they are originating from. This does not work well in all scenarios. For rays leaving from a facet with grazing angles, this can still lead to self-intersections. Further, often occurring in corners, a valid intersection might be skipped, which is shown in the left part of figure 8.14. A more robust solution is the proposed adaptive offset along the facet normal. However, this approach can place the ray through a neighboring facet, especially in geometries with sharp angles or crevices, which is shown in more detail in figure 8.14

We propose a workaround that attempts to solve the inherent problems by moving the ray away from edges where an offset along the surface normal could result in the

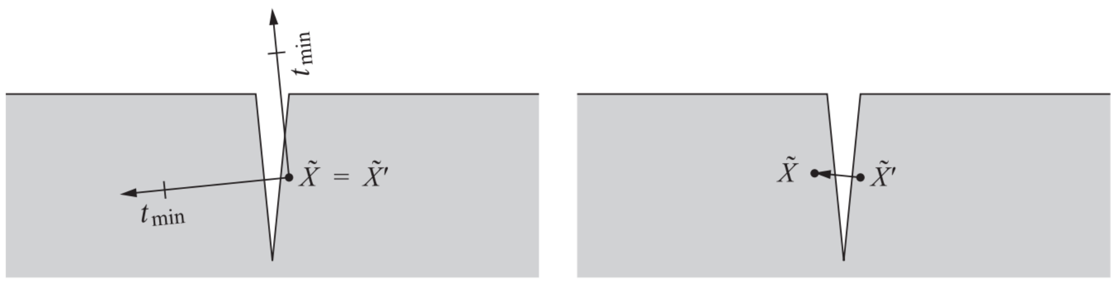


Figure 8.14: An offset along the facet normal can lead to other problems, in particular in crevices or in general where two facets are connected via a sharp angle. In this example (figure as provided by Wächter and Binder, 2019), the initial intersection $\widetilde{X}$ occurred below the surface of the geometry (grey region). Limiting the ray interval by $t_{min} = \varepsilon > 0$ works for the upper ray, but not for the lower ray (left). For the adaptive offset, the ray origin for the next ray $\widetilde{X}'$ might be moved onto a neighboring facet (right).

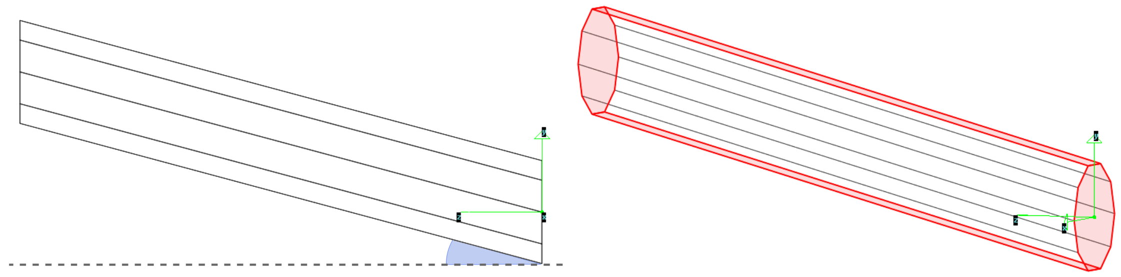
ray crossing into a neighboring facet.

8.4.1 Offset to center

In scenarios, where the adaptive offset along the normal might pose other difficulties, we suggest an additional offset strategy, which we call Neighbour Aware Offset (NAO). The adaptive offset primarily causes problems for ray locations that are close to edges, where the angle between two adjacent polygons is acute: $\alpha < 90°$. First, to identify such facets, we deploy an algorithm as proposed in section 5.4 to determine neighbor relations within a geometry. The algorithm labels two facets as `neighbor`, when they share a common edge.[12] Further, we keep track of the angle between two neighboring facets. For the GPU kernel we add the corresponding labels only for acute angles in a neighbourhood to decide in runtime, whether an additional offset should be applied to retrieve a corrected particle's origin. After applying the adaptive offset, the goal of our method is to move the ray away from edges. This strategy utilizes the geometric center of a polygon as the offset point.

We design a prototype for this offset using a modified version of the vacuum pipe geometry. The pipe is tilted by an angle α, so that the inlet and outlet remain axis-parallel. The oblique prisma that is generated by tilting with an angle $\alpha = 15$ is shown in figure 8.15. Utilising this variable test case, we develop the offset with

[12]This approach was initially chosen as it circumvents the need to label an edge. As our method applies an offset using barycentric coordinates, we would have to create an additional relation between edge and barycentric coordinates first.



(a) Tilted prism: $\alpha = 15$ (b) Highlighted acute angles

Figure 8.15: Left: An auto generated test case that shows the sample vacuum pipe test geometry tilted by an angle of $\alpha = 15°$ highlighted in blue. Right: The same example, where the *neighbor* facets are highlighted in red.

the following constraints. The offset should only be applied to rays that are located close to an edge forming a sharp angle with a neighbouring facet. The magnitude of the offset should be minimal. The computational overhead should be marginal. By default, the design targets to fulfill the last constraint by utilizing more device memory.

To fulfill the first constraint, we consider only facets with a label `neighbor`. To verify the locality of a ray on a facet, it is straightforward to use barycentric coordinates opposed to 3d world coordinates. For the latter, we would have to compute a relative position first against all facet edges. Barycentric coordinates incorporate this information by design. They are elaborated in great detail in chapter 8.3.2.1. If one of the barycentric coordinates is close to zero $\{u \approx 0 \vee v \approx 0 \vee w \approx 0\}$ then the position of the point is close to one of the triangle's edges. In a first attempt, we apply an offset independent of the edge's position. Ideally, not only facets but also individual edges should be labelled. Theoretically, this will improve the precision of the results.

We aim to empirically determine the most reasonable values for the offsets, thereby keeping their magnitudes minimal. The adaptive offset along the facet normal is crucial in this context. In our design, it serves as the first step to prevent leaks caused by floating point errors. Our methodology involves evaluating the instances of misses (`miss`) in relation to barycentric coordinates. Specifically, we examine tilted prisms with varying angles and generate binned values by counting the number of misses. These are then attributed to the smallest dimension of the barycentric coordinates (either u, v, or w). The occurance of misses, in relation to the barycentric coordinates, is illustrated in figure 8.16a. Each data point in this figure represents a binned value across an equidistant range. We find that misses already occur at barycentric coordinates ranging in magnitudes from 10^{-7} to 10^{-4}. For large acute angles, more misses occur for larger coordinates compared to tilted angles with smaller angles.

Based on these insights, we derive a scaling factor $\sigma_{bary} = [0, 1]$ that is chosen based on the proximity of a barycentric coordinate to a triangle's edge. Proximity is indicated when a barycentric coordinate u, v, w is close to zero, suggesting that the point is near one of the triangle's edges. To apply a significant weight to barycentric coordinates near the edge ($b \approx 0$), where b is a barycentric coordinate u, v, w, we use an exponential weighting function:

$$\sigma_{bary}(b) = e^{-n \cdot b} \,. \tag{8.13}$$

Here, n is selected to be 1000, ensuring that values up to $x \leq 10^{-5}$ receive sufficient weight, while values above $x > 10^{-4}$ are less impacted. This scaling factor is then applied on a facet-specific offset factor, whose values we derived empirically as elaborated in the subsequent chapter.

The proposed approach allows for a targeted application of offsets, especially near the edges of a triangle. By empirically determining the most effective offset values and applying them based on barycentric coordinates, we aim to minimize misses and enhance the precision of our ray tracing results. The empirical findings and the concrete implemenation of this approach are further elaborated.

8.4.2 Classification

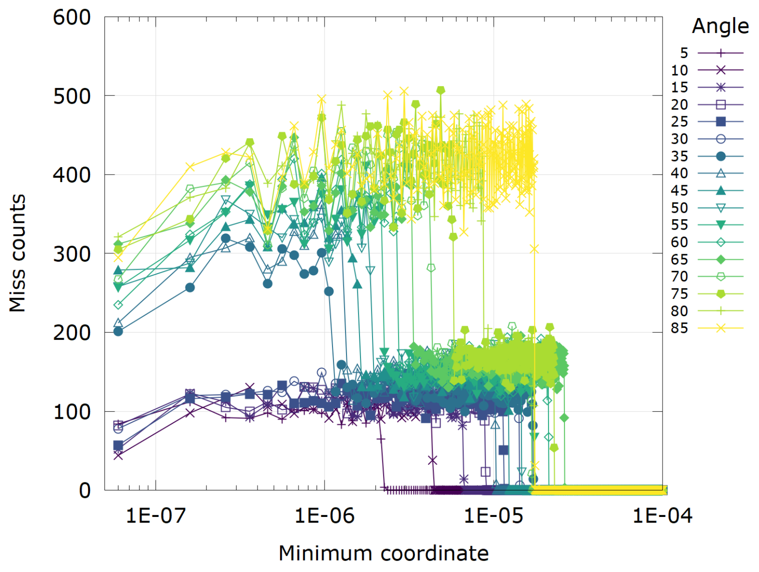
We aim to find an offset value, that corresponds well for the inherent angle of a facet defining its neighbor relations. Using equation (8.13) to compute a adequate offset towards the center:

$$\sigma_c(b) = \sigma_{facet} \cdot \sigma_{bary}(b) \,, \tag{8.14}$$

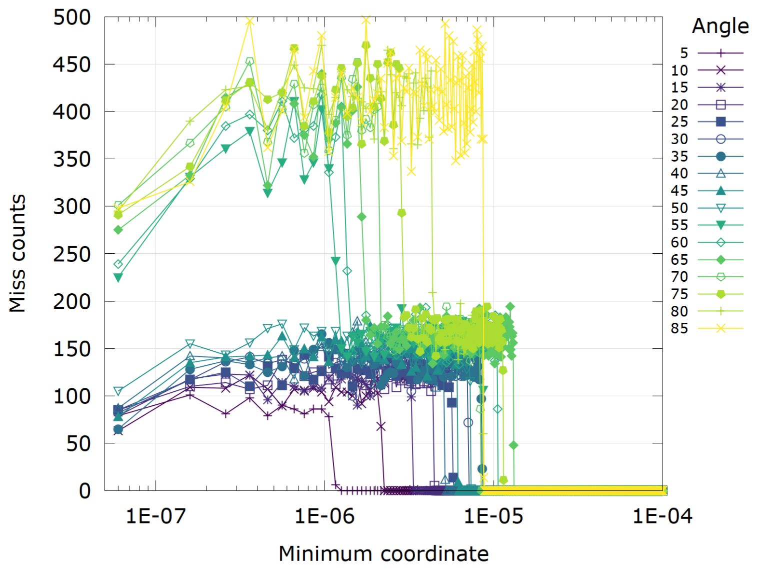
we utilize an empirically derived offset factor, σ_{facet}. For an empirical determination such an offset for each angle, we have to find minimal offset factor for which no misses occur.

Finding suitable parameters for the offsets that work for most scenarios involves a large search space, and it is not practical to manually test all possible combinations of parameters. Therefore, we need to translate this problem to an optimisation problem and use an auto-tuning framework to solve it. To achieve this, we will use the open-source software OpenTuner(Ansel et al., 2014). OpenTuner is a general-purpose auto-tuning framework that uses machine learning techniques to guide the search process. These techniques reduce the number of experiments that need to be performed to find optimized parameters. This greatly enhances the efficiency and accuracy of the tuning process by finding potential optima with a finite number of trials.

The optimization process starts with generating a test bench that covers all angles between $\alpha = [0, 90]$. For each angle, we generate an oblique prism $\mathcal{P}_\alpha$, which is a modified test problem of the standard cylindrical tube (see chapter 4.3) with a L/R

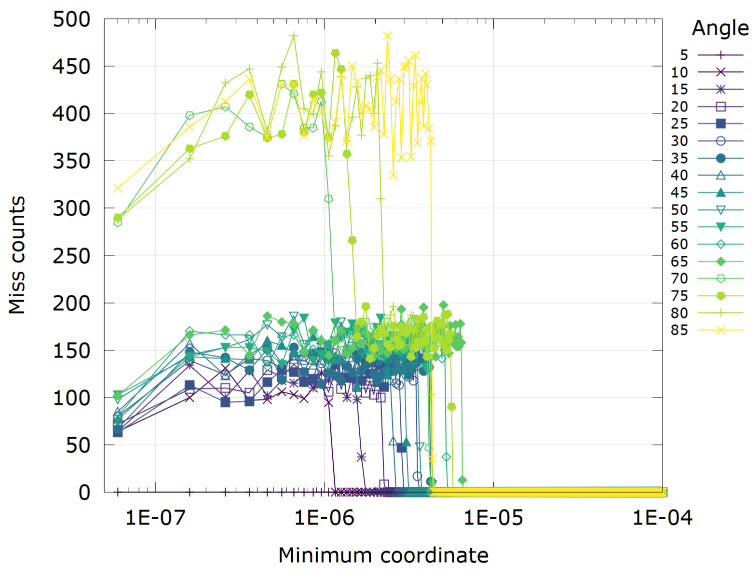


(a) No offset

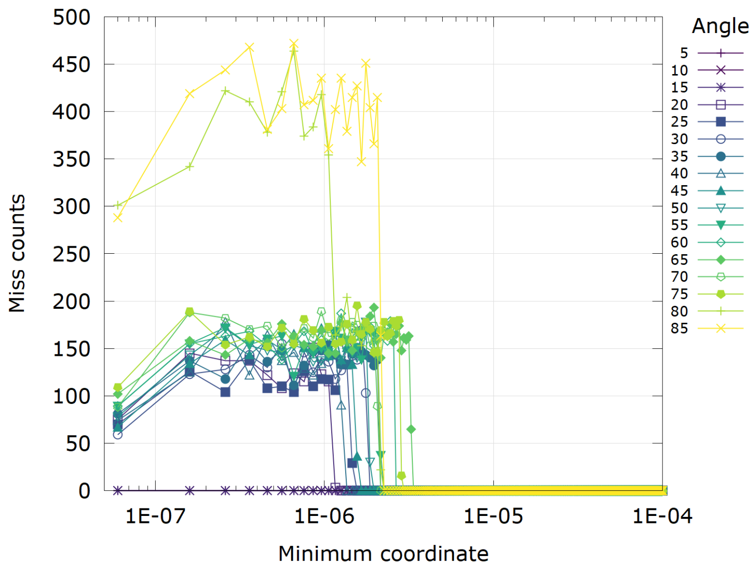


(b) Scaling factor $\sigma_{bary} = 100000$

Figure 8.16: Binned miss counts for the corresponding smallest barycentric values. For all cases, we see similar patterns. Misses occur approximately in the region from 10^{-7} to 10^{-4}. Each plot shows misses for different scaling factors. Stronger scaling factors for the offset reduce the number of missed counts, making them appear less for larger coordinate values. Figure 8.16a shows the counts without any scaling factor.



(a) Scaling factor $\sigma_{bary} = 200000$



(b) Scaling factor $\sigma_{bary} = 300000$

Figure 8.17: Binned miss counts for the corresponding smallest barycentric values. For all cases we see similar patterns. Misses occur approximately in the region from 10^{-7} to 10^{-4}. Each plot shows misses for different scaling factors. Stronger scaling factors for the offset reduce the amount of miss counts, making them appear less for larger coordinate values. Figure 8.17b shows strong reductions of occured misses, reducing them completely for some angles.

ratio of 10 and 10 side facets. The lateral facets of the prism are not aligned to the ground coordinates (in Molflow by the x- and z-coordinates), but instead tilt by an angle α. For each $\mathcal{P}_\alpha$, we search for scaling factors for the adaptive offset towards the normal σ_n and the offset towards the center σ_c that lead to ideally no leaks and minimal offset scales to prevent amplifying arithmetic errors. The optimization process returns the parameters for the offset that lead to no leaks for the particular input geometry. It is important to note that the optimization process is computationally expensive and may require significant computational resources, due to the large search space. Therefore, it is essential to carefully choose the search space and the optimisation technique to balance between the accuracy of the optimisation results and the computational cost of the process. We use OpenTuner's `AUCBanditMetaTechniqueA` technique, which deploys a set of different techniques changing them randomly after several attempts to prevent getting stuck in local extrema. In Molflow's GPU kernel, leaks are accumulated individually per originated facet. This gives us the possibility to only consider leaks during the auto-tuning experiment, which are related to the angle of interest. With Molflow's test case generation engine which utilizes 10 side facets, only the facets for the inlet and outlet and two side facets have to be taken into account as they contain an acute angle, which is highlighted in figure 8.15b. For each angle, we run 30 simulations with 150.000.000 desorptions each.

The corresponding offsets that were found using this approach using tilted vacuum pipes $\mathcal{P}_\alpha$ for varying angles α with a L/R ratio of 10 and 10 side facets are listed in table 8.4. In the table we can see, that the scaling factors increase quickly to $\alpha = 25$. For greater values, they seem to find a plateau for an upper bound. We try to determine an appropriate function in order to interpolate values also for angle values in between the samples. In total, we create three independent data sets based on the same approach. We fit a polynomial function

$$f(x) = a + b \cdot x + c \cdot x^2 + d \cdot x^3 \tag{8.15}$$

for all data sets to create a function from which we potentially can look up scaling factors. Polynoms served as an initial choice to approximate the relationship between the tilt angle α and the scaling factors due to their flexibility in modelling complex, non-linear relationships. In figure 8.18 we show the results for the three data sets. Data sets 1 and 2 show scaling factors for angles $[5, 85]$ in increments of 5. Data set 3 shows angles $[5, 85]$ for increments of 1. For the polygons fitting all three data sets we see a comparable trend. Which is why we use the polynom (8.15) fitting the data shown in table 8.4 to find scaling factors σ_{facet} for all acute angles.

The full algorithm using these values is depicted in algorithm 11. An offset is applied only if at least one of the barycentric coordinates falls below a pre-established upper limit, as identified empirically. This targeted application ensures the accuracy

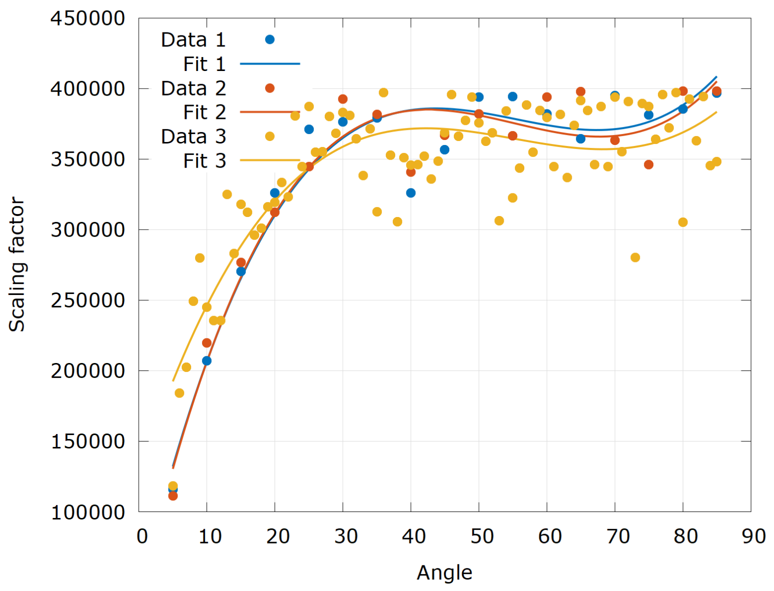


Figure 8.18: Scaling factors $\sigma_{facet} = s$ (Y-axis) found for tilted vacuum chambers with varying angles α (X-axis). Data sets 1 and 2 show values for $[5, 85]$ in increments of 5. Data set 3 shows $[5, 85]$ in increments of 1. Scaling factors increase quickly, but seem to reach an upper limit at $\alpha = 25$.

Table 8.4: Parameters for the offset to center σ_c as found via auto-tuning for tilted cylindrical tubes with varying angles α approximated with 10 side facets, where offsets were found to minimize the number of leaks and the offset itself, where each evaluation run has been done with 2×10^8 desorbed particles.

$\boldsymbol{\alpha}$	5	10	15	20	25	30
$\boldsymbol{\sigma}_{facet}$	115995	207282	270278	326098	371117	376498

$\boldsymbol{\alpha}$	35	40	45	50	55	60
$\boldsymbol{\sigma}_{facet}$	379097	326106	356732	393753	394292	381889

$\boldsymbol{\alpha}$	65	70	75	80	85
$\boldsymbol{\sigma}_{facet}$	364455	394851	381238	385572	396916

of our results. In our prototype algorithm, offsets are applied specifically for these values.[13] Moreover, in line 10 (ff.) the algorithm ensures that the value is clamped to

[13]To align with SIMD properties, the application of the offset can be strictly controlled by the individual scaling factor (σ_{bary}) for each coordinate. Which we neglected in our experiments.

a fixed constant, thereby also offsetting zero values effectively. We utilize the same integer (line 11 transformation as for the adaptive offset along the normal, which we utilize for precision reasons. We need to ensure, that the barycentric condition ($u + v + w = 1$) remains true, which is why normalize the barycentric coordinates at the end in line 22.

Algorithm 11 Offset to center using a calculation based on barycentric coordinates. The offset is applied for points, which have been offset along the facet normal.

```
 1: function OFFSET_TO_CENTER_BARY($\boldsymbol{b}, \boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C}, \boldsymbol{p}$, poly)
 2:     $c \leftarrow 1.0/3.0$                                          ▷ Barycentric center
 3:     $\mathrm{l}_{up} \leftarrow 5 \times 10^{-4}$
 4:     scale ← 256.0
 5:     $\boldsymbol{t} \leftarrow$ make_float3($\boldsymbol{b}.x, \boldsymbol{b}.y, 1.0 - \boldsymbol{b}.x - \boldsymbol{b}.y$)       ▷ Initialize barycentrics
 6:     $\sigma_{facet} \leftarrow$ poly.offset                          ▷ Load interpolated offset value
 7:     $\sigma_{bary} \leftarrow \exp(-1000 \cdot \boldsymbol{t})$                              ▷ Apply (8.13)
 8:     $\sigma_c \leftarrow \sigma_{facet} \cdot \sigma_{bary}$                                  ▷ Apply (8.14)
                              ▷ Adjust the barycentric coordinates if close to edge
 9:     if $u < \mathrm{l}_{up}$ then
10:         var ← $\max(u, 10^{-6}) + \mathrm{int}(\mathrm{scale} \cdot \sigma_c \cdot c)$
11:         $\boldsymbol{t}.x \leftarrow$ int_as_float(float_as_int(var))
12:     end if
13:     if $v < \mathrm{l}_{up}$ then
14:         var ← $\max(v, 10^{-6}) + \mathrm{int}(\mathrm{scale} \cdot \sigma_c \cdot c)$
15:         $\boldsymbol{t}.y \leftarrow$ int_as_float(float_as_int(var))
16:     end if
17:     if $w < \mathrm{l}_{up}$ then
18:         var ← $\max(w, 10^{-6}) + \mathrm{int}(\mathrm{scale} \cdot \sigma_c \cdot c)$
19:         $\boldsymbol{t}.z \leftarrow$ int_as_float(float_as_int(var))
20:     end if
        ▷ Normalize the transposed barycentric coordinate
21:     $S \leftarrow \boldsymbol{t}.x + \boldsymbol{t}.y + \boldsymbol{t}.z$
22:     $\boldsymbol{t} \leftarrow \boldsymbol{t}/S$
23:     return $\boldsymbol{t}$
24: end function
```

8.4.3 Conclusion

Given the parameter values found using OpenTuner, we were able to provide an algorithm that is capable of minimising the effects arising from the application of an adaptive offset for critical parts of a geometry. The values have been integrated as a scaling offset σ_{facet} and is evaluated by acquiring a value based on (8.15). The parameters a, b, c, d have been determined by fitting the function with the empirically found values that are listed in table 8.4. We define the parameters values as $a = 40473$, $b = 20198.9$, $c = -381.51$ and $d = 2.29206$. By running the simulations with the defined offset scales, we could completely remove the occurrence of leaks. In table 8.5 we show the results for multiple runs for 10^9 desorbed particles. Comparing the values for the computed transmission probabilities for the CPU kernel w_{CPU} and the GPU w_{GPU}, we see that the relative error for all cases has magnitude 10^{-3}. The initial kernel, that was performant on cylindrical vacuum tubes (results summarized in table 8.3) that didn't show any sharp angles, was more accurate using only the adaptive offset along the normal. In these scenarios, the offset towards the center did not have to be applied, as no acute angles exist. The relative difference between transmission probabilities w_{CPU} and w_{GPU}, calculated using the CPU and the GPU kernel respectively, is in average in the magnitude of 10^{-4}.

While the initial GPU kernel was usable on simple geometries such as cylindrical tubes, a modification was necessary for more complex geometries. We provided a possible solution to achieve better results on more complex geometries, by mitigating the effects of particle leaks. Our method is capable of applying the offset only for specific parts by flagging corresponding facets as a pre-computation step, by evaluating facet relationships. Although the determined parameters are only quasi-ideal due to the enormous search space, they were able to remove particle misses completely when used for a more complex test case. For a more general approach, the parameter space has to be extended, because the parameters are not scale-invariant. Here, the oblique prism generator could be extended to further include a scaling parameter. Another potential solution could be the use of a normalized coordinate system for which one set of parameters could be used. A downside of the utilisation of barycentric coordinates is, that the offsets achieved are not scale-invariant. This is because barycentric coordinates are in the range $[0, 1]$ and give the location in the triangle in percent. Because of this trait, the scale information is inherently lost. The same offset applied to barycentric coordinates of different facets may result in different magnitudes of differences in absolute world coordinates. An idea to find a truly portable solution, that can be applied on other geometries, is to find a scaling factor that is scale invariant, e.g. one that can be applied directly on real world coordinates. This, on the other hand, might impose other challenges for a robust solution. Further, a more ideal solution would need to apply an offset such as σ_{facet} not per facet, but per edge. Due to this generalization, it is likely that the offset is applied in many scenarios, where the actual location is not close to an acute angle, further enhancing the error in the simulation. Lastly, we could see a stronger negative impact on the precision of the results compared to the usage of only the adaptive offset and axis-aligned vacuum chambers pipes. It has to be investigated in another study whether the reduction in precision makes the GPU kernel impractical.

Table 8.5: Simulations for the tilted prism geometry $\mathcal{P}_\alpha$ for $\alpha = [5, 85]$ using the CPU and the GPU kernel for 10^9 desorptions. The CPU algorithm is using 64 bit calculations. The GPU algorithm is using 32 bit calculations and the offset towards the center. The transmission probability $w_{CPU|GPU}$ is evaluated as the key value. The absolute and relative differences between the two calculated values is given. Here, the CPU algorithm serves as gold standard.

Angle	w_{CPU}	w_{GPU}	Δ_{abs}	Δ_{rel}
5	0.185266140	0.186547331	0.001281191	0.006891577
10	0.182628734	0.183906607	0.001277873	0.006972716
15	0.178254740	0.179590804	0.001336064	0.007467266
20	0.172230853	0.173549358	0.001318505	0.007626263
25	0.164628610	0.165896040	0.001267430	0.007669201
30	0.155327253	0.156652590	0.001325337	0.008496300
35	0.144678630	0.145915447	0.001236817	0.008512333
40	0.132541272	0.133848067	0.001306795	0.009811164
45	0.119372791	0.120476158	0.001103367	0.009200514
50	0.104987216	0.106007837	0.001020621	0.009674363
55	0.089908290	0.090696999	0.000788709	0.008734058
60	0.074243976	0.074917452	0.000673476	0.009030160
65	0.058371200	0.059028755	0.000657555	0.011201963
70	0.042846960	0.043376352	0.000529392	0.012279554
75	0.028118795	0.028290723	0.000171928	0.006095701
80	0.015039693	0.015020494	0.000019199	0.001277378
85	0.004848339	0.004763459	0.000084880	0.017661681

Further, the offset could be optimized to work even more selectively in those cases where a particle miss will occur and only with such a strong offset magnitude, that the negative effects on the accuracy will be minimized.

9 Conclusion and Future Work

This thesis presents a comprehensive exploration of multiple aspects of Molflow, including algorithmic improvements for preprocessing, simulation enhancements on CPU, advanced ray tracing techniques, and GPU/OptiX kernel development. This thesis has explored various aspects of accelerating Molflow's Test Particle Monte Carlo algorithm, a vital tool for simulating the behavior of particles in vacuum systems. We have made substantial strides in improving Molflow's performance and scalability, thanks to a revamped ray-tracing engine and enhanced support for distributed computing using MPI. We have investigated and developed new acceleration data structures tailored to Molflow's specific traits. Additionally, improvements have been made to the performance of preprocessing algorithms and in developing tools to aid users in data analysis. We developed a GPU kernel, based on NVIDIA OptiX, that leverages ray tracing cores. This kernel includes an offsetting algorithm to reduce errors and mitigates various effects from single-precision calculations. Enhancements to the ray tracing engine, as well as the new CLI are part of the public releases for both Molflow and Synrad. With additional enhancements to the time-dependent algorithm and the neighbour analysis, this work concluded with Molflow 2.9.5 and Synrad 1.5.0.

The Monte Carlo kernel has been updated with an OpenMP back-end and a revamped ray tracing engine that is utilizing state of the art acceleration data structures and traversal algorithms. Based on PBRT's open source code, , we investigated the effectiveness of various BVHs and KD-trees, along with different splitting techniques and traversal algorithms. We developed statistical techniques that leverage data specific to Molflow simulations to create acceleration data structures excelling the quality of other techniques. Based on the assumption, that KD-trees are more suitable than BVHs due to their advantages for static geometries, we developed an enhanced rope traversal algorithm making use of previous hit locations. While the algorithm proved noteworthy in some scenarios, the results for the implementation of all KD-tree-based algorithms have been worse on average than those for BVHs. In a targeted study featuring an optimized KD-tree implementation and enhancements to the newly developed traversal algorithm, which is based on the concept of rope-traversal, the strengths of these techniques should be revalidated. Further a method to dynamically select the best acceleration data structure should be developed, which could potentially work on pre-evaluating a set of geometry characteristics such as those proposed.

At the time of finalizing this thesis, the iterative simulation algorithm had been further developed. Techniques such as the convergence-based stopping criteria and the statistic-based construction algorithms were developed specifically for iterative simulations. When fully integrated, their usefulness can be validated. The stopping criterion promises better intermediate results, that will reduce the error propagation for the next iteration. Employing acceleration data structures in combination with splitting heuristics, like the RDH or HRH, based on existing statistics could enhance the performance of ray tracing queries.

We developed a GPU kernel powered by NVIDIA's OptiX 7 API to utilize the novel ray tracing cores found in recent GPU architectures. We analyzed different techniques to design the algorithm to achieve peak performance for the given geometry, while also keeping memory into account. Our design achieved major speed-ups for cylindrical vacuum tube geometries on budget hardware (63×–175×) as well as on high end hardware (30×–50×) when comparing GPU against CPU simulations, without heavy influence on the precision. In the GPU study of this thesis, we investigated issues that can arise for physical simulations as a trade-off for a highly performant GPU kernel fully utilizing new RTX hardware. The adaptive offset mitigates effects such as the displaced ray origins, but it does not solve all issues by itself. To counter these effects, we implement an adaptive offset to the center. Using the properties of barycentric coordinates, only rays that are close to a triangles edge will be offset. The offset accounts for neighboring facets that join with a sharp angle. An empirical study was conducted to find scaling values that are applied depending on the corresponding angles. The proposed method extends the range of geometries, the GPU kernel can be used for. The robustness of the method still has to be validated for other facet properties and more complex geometries, that are more prone to single precision errors. A detailed analysis and design study for such problems and the implementation of Molflow's full feature set are left for future work.

With the work of this thesis, the open-source project has been improved in many different ways. CI/CD integration with a rich testing infrastructure and the refactored code base gives developers more reliable tools to enhance the individual aspects of the code as well as to add new functionality. Major performance improvements have been achieved not only on the simulation side. Users can leverage the major simulation performance even on dedicated compute resources due to the newly developed command line interface `MolflowCLI`. The developed GPU kernel has proven to be highly performant for a selected set of test cases. Running simulations on dedicated ray tracing hardware is an anticipated future. But first, all functions of Molflow's Test Particle Monte Carlo code have to be implemented and a robust solution for the underlying issues has be identified and deployed following a thorough validation of the kernel. The lessons from the investigations and the strengths and limitations of the developed techniques are valuable for many other physical simulation codes, as the traits of the Monte Carlo algorithms are similar among them.

Bibliography

[Aas20] Jordan Aasman. "Laser Transport System Vacuum Simulations and LED Atom Tracker". In: (Aug. 2020). DOI: 10.2172/1661680. URL: https://www.osti.gov/biblio/1661680.

[Ady16] Marton Ady. "Monte Carlo simulations of ultra high vacuum and synchrotron radiation for particle accelerators". Presented 03 May 2016. May 2016. URL: https://cds.cern.ch/record/2157666.

[AHH08] Tomas Akenine-Möller, Eric Haines, and Naty Hoffman. *Real-Time Rendering*. 3rd ed. A K Peters/CRC Press, 2008. DOI: 10.1201/9781315365459. URL: https://doi.org/10.1201/9781315365459.

[AK89] James Arvo and David B. Kirk. "An introduction to ray tracing". In: 1989.

[AKL13] Timo Aila, Tero Karras, and Samuli Laine. "On quality metrics of bounding volume hierarchies". In: July 2013, pp. 101–107. DOI: 10.1145/2492045.2492056.

[Alr+21] John M. Alred et al. "Designing a Decontamination Solution for the Low-Earth-Orbit, Cryogenic SPHEREx Mission". In: *2021 IEEE Aerospace Conference (50100)*. 2021, pp. 1–8. DOI: 10.1109/AERO50100.2021.9438321.

[Ans+14] Jason Ansel et al. "OpenTuner: An Extensible Framework for Program Autotuning". In: *International Conference on Parallel Architectures and Compilation Techniques (PACT)*. Edmonton, Canada, Aug. 2014. URL: http://groups.csail.mit.edu/commit/papers/2014/ansel-pact14-opentuner.pdf.

[Ata07] Mustafa Y. Ata. "A convergence criterion for the Monte Carlo estimates". In: *Simulation Modelling Practice and Theory* 15 (Mar. 2007), pp. 237–246. DOI: 10.1016/j.simpat.2006.12.002.

[Bäh+22] Pascal R Bähr et al. "Development of a hardware-accelerated simulation kernel for ultra-high vacuum with Nvidia RTX GPUs". In: *The International Journal of High Performance Computing Applications* 36.2 (2022), pp. 141–152. DOI: 10.1177/10943420211056654. eprint: https

://doi.org/10.1177/10943420211056654. URL: https://doi.org/10.1177/10943420211056654.

[BH09] Jiri Bittner and Vlastimil Havran. “RDH: Ray distribution heuristics for construction of spatial data structures”. In: (May 2009). DOI: 10.1145/1980462.1980475.

[Bir76] G. A. Bird. “Molecular gas dynamics”. In: *NASA STI/Recon Technical Report A* 76 (Jan. 1976), p. 40225.

[Bly20] Simon Blyth. “Meeting the challenge of JUNO simulation with Opticks: GPU optical photon acceleration via NVIDIA® OptiXTM”. In: *EPJ Web Conf.* 245 (2020), p. 11003. DOI: 10.1051/epjconf/202024511003.

[Boo15] Thomas Booth. *Intuition and Variance Reduction in Monte Carlo Simulations.* Nov. 2015. DOI: 10.13140/RG.2.1.1790.1522.

[Bro06] Forrest B. Brown. “On the Use of Shannon Entropy of the Fission Distribution for Assessing Convergence of Monte Carlo Criticality Calculations”. In: 2006.

[Bru20] D. vom Bruch. “Real-time data processing with GPUs in high energy physics”. In: *Journal of Instrumentation* 15.06 (June 2020), pp. C06010–C06010. DOI: 10.1088/1748-0221/15/06/c06010. URL: https://doi.org/10.1088/1748-0221/15/06/c06010.

[Bur+18] Philip N. Burrows et al. “The Compact Linear Collider (CLIC) - 2018 Summary Report”. In: *CERN Yellow Reports: Monographs* 2 (2018). ISSN: 2519-8068 (Print), 2519-8076 (Online). DOI: 10.23731/CYRM-2018-002.

[CC06] P. Chiggiato and P. Costa Pinto. “Ti–Zr–V non-evaporable getter films: From development to large scale production for the Large Hadron Collider”. In: *Thin Solid Films* 515.2 (2006). Proceedings of the Eighth International Conference on Atomically Controlled Surfaces, Interfaces and Nanostructures and the Thirteenth International Congress on Thin Films, pp. 382–388. ISSN: 0040-6090. DOI: https://doi.org/10.1016/j.tsf.2005.12.218. URL: https://www.sciencedirect.com/science/article/pii/S0040609005025496.

[CGL83] Tony F. Chan, Gene H. Golub, and Randall J. LeVeque. “Algorithms for Computing the Sample Variance: Analysis and Recommendations”. In: *The American Statistician* 37.3 (1983), pp. 242–247. ISSN: 00031305. URL: http://www.jstor.org/stable/2683386 (visited on 11/11/2023).

[Coo12] Shane Cook. *CUDA Programming: A Developer's Guide to Parallel Computing with GPUs*. 1st. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2012. ISBN: 9780124159334.

[Del+34] Boris Delaunay et al. "Sur la sphere vide". In: *Izv. Akad. Nauk SSSR, Otdelenie Matematicheskii i Estestvennyka Nauk* 7.793-800 (1934), pp. 1–2.

[Den+17] Yangdong Deng et al. "Toward Real-Time Ray Tracing: A Survey on Hardware Acceleration and Microarchitecture Techniques". In: *ACM Comput. Surv.* 50.4 (Aug. 2017). ISSN: 0360-0300. DOI: 10.1145/3104067. URL: https://doi.org/10.1145/3104067.

[Dra90] Douglas J. Drake. "View-factor method for solving time-dependent radiation transport problems involving fixed surfaces with intervening, participating media". In: *Journal of Computational Physics* 87.1 (1990), pp. 73–90. ISSN: 0021-9991. DOI: https://doi.org/10.1016/0021-9991(90)90226-Q. URL: https://www.sciencedirect.com/science/article/pii/002199919090226Q.

[DRS10] J. De Loera, J. Rambau, and F. Santos. *Triangulations: Structures for Algorithms and Applications*. Algorithms and Computation in Mathematics. Springer Berlin Heidelberg, 2010. ISBN: 9783642129711. URL: https://books.google.de/books?id=SxY1Xrr12DwC.

[DS12] M.H. DeGroot and M.J. Schervish. *Probability and Statistics*. Addison-Wesley, 2012. ISBN: 9780321500465. URL: https://books.google.de/books?id=4TlEPgAACAAJ.

[FLF12] Nicolas Feltman, Minjae Lee, and Kayvon Fatahalian. "SRDH: Specializing BVH Construction and Traversal Order Using Representative Shadow Ray Sets". In: June 2012, pp. 49–55. DOI: 10.2312/EGGH/HPG12/049-055.

[Fra18] W. R. Franklin. *PNPOLY – Point Inclusion in Polygon Test*. 2018. URL: wrf.ecse.rpi.edu/Research/Short_Notes/pnpoly.html (visited on 12/08/2020).

[Fra94] W. Randolph Franklin. *PNPOLY - point inclusion in Polygon Test W. Randolph Franklin (WRF)*. 1994. URL: https://wrfranklin.org/Research/Short_Notes/pnpoly.html.

[GL03] J. Gómez-Goñi and P. Lobo. "Comparison between Monte Carlo and analytical calculation of the conductance of cylindrical and conical tubes". In: *Journal of Vacuum Science & Technology A - J VAC SCI TECHNOL A* 21 (July 2003). DOI: 10.1116/1.1568746.

[GN13] Christiaan Gribble and Alexis Naveros. “GPU Ray Tracing with Rayforce”. In: *ACM SIGGRAPH 2013 Posters*. SIGGRAPH ’13. Anaheim, California: Association for Computing Machinery, 2013. ISBN: 9781450323420. DOI: 10.1145/2503385.2503493. URL: https://doi.org/10.1145/2503385.2503493.

[GNK14] Christiaan Gribble, Alexis Naveros, and Ethan Kerzner. “Multi-Hit Ray Traversal”. In: *Journal of Computer Graphics Techniques (JCGT)* 3.1 (Feb. 2014), pp. 1–17. ISSN: 2331-7418. URL: http://jcgt.org/published/0003/01/01/.

[Hag20] Alireza Haghighat. *Monte Carlo Methods for Particle Transport*. July 2020. ISBN: 9780429198397. DOI: 10.1201/9780429198397.

[HAK23] P.L. Henriksen, M. Ady, and R. Kersevan. “Vacuum chamber conditioning and saturation simulation tool (VacuumCOST): Enabling time-dependent simulations of pressure and NEG sticking in UHV chambers”. In: *Vacuum* 212 (2023), p. 111992. ISSN: 0042-207X. DOI: https://doi.org/10.1016/j.vacuum.2023.111992. URL: https://www.sciencedirect.com/science/article/pii/S0042207X23001896.

[Has20] Md. Zahid Hasan. “The fidelity of DSMC method to analyze the aerothermodynamics of the pure continuum flow regime”. In: *International Journal of Thermofluids* 7-8 (2020), p. 100047. ISSN: 2666-2027. DOI: https://doi.org/10.1016/j.ijft.2020.100047. URL: https://www.sciencedirect.com/science/article/pii/S2666202720300343.

[Hav00] Vlastimil Havran. “Heuristic Ray Shooting Algorithms”. PhD thesis. Nov. 2000.

[HB07] Vlastimil Havran and Jiri Bittner. “Stackless Ray Traversal for kD-Trees with Sparse Boxes”. In: (Dec. 2007).

[ISP07] Thiago Ize, Peter Shirley, and Steven Parker. “Grid Creation Strategies for Efficient Ray Tracing”. In: *2007 IEEE Symposium on Interactive Ray Tracing*. 2007, pp. 27–32. DOI: 10.1109/RT.2007.4342587.

[Jen21] Noah Jens. “Monte Carlo simulations of light Estimating and comparing the photon rates of different sources at the Beam Gas Curtain monitor’s camera”. Presented 01 Jul 2021. Technische Hochschule Luebeck, 2021. URL: https://cds.cern.ch/record/2775143.

[KA13] Tero Karras and Timo Aila. “Fast Parallel Construction of High-Quality Bounding Volume Hierarchies”. In: *Proceedings of the 5th High-Performance Graphics Conference*. HPG ’13. Anaheim, California: Association for Computing Machinery, 2013, pp. 89–99. ISBN: 9781450321358. DOI: 10

.1145/2492045.2492055. URL: https://doi.org/10.1145/2492045.2492055.

[KA19] Roberto Kersevan and Marton Ady. "Recent developments of Monte-Carlo codes Molflow+ and Synrad+". In: *10th International Particle Accelerator Conference*. 2019, TUPMP037. DOI: 10.18429/JACoW-IPAC2019-TUPMP037.

[Kam+21] Junichiro Kamiya et al. "Improved vacuum system for high-power proton beam operation of the rapid cycling synchrotron". In: *Phys. Rev. Accel. Beams* 24 (8 Aug. 2021), p. 083201. DOI: 10.1103/PhysRevAccelBeams.24.083201. URL: https://link.aps.org/doi/10.1103/PhysRevAccelBeams.24.083201.

[Ker22] Roberto Kersevan. "The FCC-ee vacuum system, from conceptual to prototyping". In: *EPJ Techniques and Instrumentation* 9.1 (2022), p. 12. ISSN: 2195-7045. DOI: 10.1140/epjti/s40485-022-00087-w. URL: https://doi.org/10.1140/epjti/s40485-022-00087-w.

[Ker91] R Kersevan. "Molflow user's guide". In: *available from one of the authors (RK)* (1991).

[KFS20] Shikhar Kumar, Benoit Forget, and Kord Smith. "Stationarity Diagnostics using Functional Expansion Tallies". In: *Annals of Nuclear Energy* 143 (2020), p. 107388. ISSN: 0306-4549. DOI: https://doi.org/10.1016/j.anucene.2020.107388. URL: https://www.sciencedirect.com/science/article/pii/S0306454920300864.

[KK86] Timothy L. Kay and James T. Kajiya. "Ray Tracing Complex Scenes". In: *SIGGRAPH Comput. Graph.* 20.4 (Aug. 1986), pp. 269–278. ISSN: 0097-8930. DOI: 10.1145/15886.15916. URL: http://doi.acm.org/10.1145/15886.15916.

[Knu67] M. Knudsen. *The Cosine Law in the Kinetic Theory of Gases*. NASA technical translation. National Aeronautics and Space Administration, 1967. URL: https://books.google.de/books?id=XEAnB-_MG-kC.

[KP09] Roberto Kersevan and J.-L Pons. "Introduction to MOLFLOW+: New graphical processing unit-based Monte Carlo code for simulating molecular flows and for calculating angular coefficients in the compute unified device architecture environment". In: *Journal of Vacuum Science & Technology A: Vacuum, Surfaces, and Films* 27 (Aug. 2009), pp. 1017–1023. DOI: 10.1116/1.3153280.

[KTB11] D.P. Kroese, T. Taimre, and Z.I. Botev. *Handbook of Monte Carlo Methods*. Wiley Series in Probability and Statistics. Wiley, 2011. ISBN: 9780470177938. URL: `https://books.google.de/books?id=-j3bmyGXKUIC`.

[Lau+09] C. Lauterbach et al. "Fast BVH Construction on GPUs". In: *Computer Graphics Forum* 28.2 (2009), pp. 375–384. DOI: `https://doi.org/10.1111/j.1467-8659.2009.01377.x`. eprint: `https://onlinelibrary.wiley.com/doi/pdf/10.1111/j.1467-8659.2009.01377.x`. URL: `https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1467-8659.2009.01377.x`.

[LDG17] Zonghui Li, Yangdong Deng, and Ming Gu. "Path Compression Kd-Trees with Multi-Layer Parallel Construction a Case Study on Ray Tracing". In: *Proceedings of the 21st ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*. I3D '17. San Francisco, California: Association for Computing Machinery, 2017. ISBN: 9781450348867. DOI: `10.1145/3023368.3023382`. URL: `https://doi.org/10.1145/3023368.3023382`.

[LEc12] Pierre L'Ecuyer. "Random Number Generation". In: *Handbook of Computational Statistics: Concepts and Methods*. Ed. by James E. Gentle, Wolfgang Karl Härdle, and Yuichi Mori. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 35–71. ISBN: 978-3-642-21551-3. DOI: `10.1007/978-3-642-21551-3_3`. URL: `https://doi.org/10.1007/978-3-642-21551-3_3`.

[LTL14] Artur Lira dos Santos, Veronica Teichrieb, and Jorge Lindoso. "Review and Comparative Study of Ray Traversal Algorithms on a Modern GPU Architecture". In: June 2014.

[Maj+18] Alexander Majercik et al. "A Ray-Box Intersection Algorithm and Efficient Dynamic Voxel Rendering". In: *Journal of Computer Graphics Techniques (JCGT)* 7.3 (Sept. 2018), pp. 66–81. ISSN: 2331-7418. URL: `http://jcgt.org/published/0007/03/04/`.

[Mar03] G. Marsaglia. "Random Number Generators". In: *Journal of Modern Applied Statistical Methods* 2 (2003), pp. 2–13.

[Mar68] George Marsaglia. "Random numbers fall mainly in the planes". In: *Proc Natl Acad Sci U S A* 61.1 (1968), pp. 25–28. DOI: `10.1073/pnas.61.1.25`.

[MB90] J. David MacDonald and Kellogg S. Booth. "Heuristics for ray tracing using space subdivision". In: *The Visual Computer* 6 (1990), pp. 153–166.

[Mei+21] Daniel Meister et al. "A Survey on Bounding Volume Hierarchies for Ray Tracing". In: *Computer Graphics Forum* 40.2 (2021), pp. 683–712. DOI: https://doi.org/10.1111/cgf.142662. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/cgf.142662. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/cgf.142662.

[Nab+13] Kosuke Nabata et al. "Efficient divide-and-conquer ray tracing using ray sampling". In: July 2013, pp. 129–135. DOI: 10.1145/2492045.2492059.

[NVI18] NVIDIA. *NVIDIA Turing GPU Architecture.* White paper. WP-09183-001_v01 available at https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf. NVIDIA Corporation, 2018.

[NVI19] NVIDIA. *cuRAND Library – Programming Guide, CUDA Toolkit v11.1.1.* 2019. URL: https://docs.nvidia.com/cuda/curand (visited on 12/09/2020).

[NVI20] NVIDIA. *NVIDIA OptiX 7.2 – Programming Guide.* 2020. URL: https://raytracing-docs.nvidia.com/optix7/guide/index.html (visited on 12/07/2020).

[NVI23] NVIDIA. *CUDA Refresher: CUDA Programming Model.* https://developer.nvidia.com/blog/cuda-refresher-cuda-programming-model/. Accessed: 2023, Dec 03. 2023.

[ONe14] Melissa E. O'Neill. *PCG: A Family of Simple Fast Space-Efficient Statistically Good Algorithms for Random Number Generation.* Tech. rep. HMC-CS-2014-0905. Claremont, CA: Harvey Mudd College, Sept. 2014.

[Osa+02] Robert Osada et al. "Shape Distributions". In: *ACM Transactions on Graphics* 21.4 (Oct. 2002), pp. 807–832.

[Par+10] Steven G. Parker et al. "OptiX: A General Purpose Ray Tracing Engine". In: *ACM Trans. Graph.* 29.4 (July 2010). ISSN: 0730-0301. DOI: 10.1145/1778765.1778803. URL: https://doi.org/10.1145/1778765.1778803.

[PIA78] Yehoshua Perl, Alon Itai, and Haim Avni. "Interpolation search—a log log N search". In: *Communications of the ACM* 21 (July 1978), pp. 550–553. DOI: 10.1145/359545.359557.

[PJH17] Matt Pharr, Wenzel Jakob, and Greg Humphreys, eds. *Physically Based Rendering: From Theory to Implementation.* Third Edition. Boston: Morgan Kaufmann, 2017, IFC. ISBN: 978-0-12-800645-0. DOI: `https://doi.org/10.1016/B978-0-12-800645-0.50030-0`. URL: `https://www.sciencedirect.com/book/9780128006450/physically-based-rendering`.

[PL10] Jacopo Pantaleoni and David Luebke. “HLBVH: Hierarchical LBVH Construction for Real-Time Ray Tracing of Dynamic Geometry”. In: *High Performance Graphics.* Ed. by Michael Doggett, Samuli Laine, and Warren Hunt. The Eurographics Association, 2010. ISBN: 978-3-905674-26-2. DOI: `10.2312/EGGH/HPG10/087-095`.

[Pop+07] Stefan Popov et al. “Stackless KD-Tree Traversal for High Performance GPU Ray Tracing”. In: *Cohen-Or, Daniel; Slavik, Pavel: Eurographics 2007, Blackwell, 415-424 (2007)* 26 (Sept. 2007). DOI: `10.1111/j.1467-8659.2007.01064.x`.

[Qi+20] Pian Qi et al. “A parallel solution to finding nodal neighbors in generic meshes”. In: *MethodsX* 7 (2020), p. 100954. ISSN: 2215-0161. DOI: `https://doi.org/10.1016/j.mex.2020.100954`. URL: `https://www.sciencedirect.com/science/article/pii/S2215016120301746`.

[RFS22] Valentina Ricchiuti, Daniel Fugett, and Carlos Soares. “Modeling of contamination vent path for outgassing components underneath thermal blankets on Europa Clipper”. In: *Space Systems Contamination: Prediction, Control, Performance 2022.* Ed. by Carlos E. Soares, Eve M. Wooldridge, and Bruce A. Matheson. Vol. 12224. Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series. Oct. 2022, 122240K, 122240K. DOI: `10.1117/12.2633991`.

[Rom+15] Paul K. Romano et al. “OpenMC: A state-of-the-art Monte Carlo code for research and development”. In: *Annals of Nuclear Energy* 82 (2015). Joint International Conference on Supercomputing in Nuclear Applications and Monte Carlo 2013, SNA + MC 2013. Pluri- and Trans-disciplinarity, Towards New Modeling and Numerical Simulation Paradigms, pp. 90–97. ISSN: 0306-4549. DOI: `https://doi.org/10.1016/j.anucene.2014.07.048`. URL: `http://www.sciencedirect.com/science/article/pii/S030645491400379X`.

[Ros04] A Rossi. *VASCO (VAcuum Stability COde): multi-gas code to calculate gas density profile in a UHV system.* Tech. rep. Geneva: CERN, 2004. URL: `https://cds.cern.ch/record/728512`.

[Sak80] G. L. Saksaganskii. “Molecular flows in complex vacuum structures”. In: *Moscow Atomizdat* (Jan. 1980).

[Sch08] Stefan Schirra. “How Reliable Are Practical Point-in-Polygon Strategies?” In: *Algorithms - ESA 2008*. Ed. by Dan Halperin and Kurt Mehlhorn. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 744–755. ISBN: 978-3-540-87744-8.

[SDS15] Doug Schouten, Adam DeAbreu, and Bernd Stelzer. “GPUs for Higgs boson data analysis at the LHC using the Matrix Element Method”. In: *GPU Computing in High-Energy Physics*. 2015, pp. 109–118. DOI: 10.3204/DESY-PROC-2014-05/20.

[SH12] Diego Song and Carlos Hernandez-Garcia. “Simulating Pressure Profiles for the Free-Electron Laser Photoemission Gun Using Molflow+”. In: *APS Division of Nuclear Physics Meeting Abstracts*. APS Meeting Abstracts. Oct. 2012, EA.095, EA.095.

[Shn78] Ben Shneiderman. “Jump searching: a fast sequential search technique”. In: *Commun. ACM* 21 (1978), pp. 831–834. URL: https://api.semanticscholar.org/CorpusID:17838241.

[Shr18] Patrick C Shriwise. *Geometry Query Optimizations in CAD-based Tessellations for Monte Carlo Radiation Transport*. The University of Wisconsin-Madison, 2018.

[SM09] Peter Shirley and Steve Marschner. *Fundamentals of Computer Graphics*. 3rd. Natick, MA, USA: A. K. Peters, Ltd., 2009. ISBN: 1568814690, 9781568814698.

[SM19] Justin Salmon and Simon McIntosh-Smith. “Exploiting Hardware-Accelerated Ray Tracing for Monte Carlo Particle Transport with OpenMC”. In: Nov. 2019, pp. 19–29. DOI: 10.1109/PMBS49563.2019.00008.

[SSK07] Maxim Shevtsov, Alexei Soupikov, and Alexander Kapustin. “Highly Parallel Fast KD-tree Construction for Interactive Ray Tracing of Dynamic Scenes”. In: *Comput. Graph. Forum* 26 (Sept. 2007), pp. 395–404. DOI: 10.1111/j.1467-8659.2007.01062.x.

[Sue96] Yusuke Suetsugu. “Application of the Monte Carlo method to pressure calculation”. In: *Journal of Vacuum Science & Technology A* 14.1 (Jan. 1996), pp. 245–250. ISSN: 0734-2101. DOI: 10.1116/1.579927. eprint: https://pubs.aip.org/avs/jva/article-pdf/14/1/245/11777757/245_1_online.pdf. URL: https://doi.org/10.1116/1.579927.

[Sun21] Daniel Sunday. “Practical Geometry Algorithms: With C++ Code”. In: *KDP Print US* (2021).

[SWM21] Peter Shirley, Ingo Wald, and Adam Marrs. “Ray Axis-Aligned Bounding Box Intersection”. In: *Ray Tracing Gems II: Next Generation Real-Time Rendering with DXR, Vulkan, and OptiX*. Ed. by Adam Marrs, Peter Shirley, and Ingo Wald. Berkeley, CA: Apress, 2021, pp. 37–39. ISBN: 978-1-4842-7185-8. DOI: `10.1007/978-1-4842-7185-8_2`. URL: `https://doi.org/10.1007/978-1-4842-7185-8_2`.

[Ven+19] R. Veness et al. “Development of a Beam-Gas Curtain Profile Monitor for the High Luminosity Upgrade of the LHC”. In: *Proc. 7th International Beam Instrumentation Conference (IBIC’18), Shanghai, China, 09-13 September 2018* (Shanghai, China). International Beam Instrumentation Conference 7. https://doi.org/10.18429/JACoW-IBIC2018-WEPB16. Geneva, Switzerland: JACoW Publishing, Jan. 2019, pp. 472–476. ISBN: 978-3-95450-201-1. DOI: `doi:10.18429/JACoW-IBIC2018-WEPB16`. URL: `http://jacow.org/ibic2018/papers/wepb16.pdf`.

[VVS+19] V.V.Sanzharov et al. “Examination of the Nvidia RTX”. In: Nov. 2019, pp. 7–12. DOI: `10.30987/graphicon-2019-2-7-12`.

[Wal+14] Ingo Wald, Sven Woop, et al. “Embree: A Kernel Framework for Efficient CPU Ray Tracing”. In: *ACM Trans. Graph.* 33.4 (July 2014). ISSN: 0730-0301. DOI: `10.1145/2601097.2601199`. URL: `https://doi.org/10.1145/2601097.2601199`.

[Wal+19] Ingo Wald, Will Usher, et al. “RTX Beyond Ray Tracing: Exploring the Use of Hardware Ray Tracing Cores for Tet-Mesh Point Location”. In: *High-Performance Graphics - Short Papers*. 2019. DOI: `10.2312/hpg.20191189`.

[Wal21] Ingo Wald. “Improving Numerical Precision in Intersection Programs”. In: *Ray Tracing Gems II: Next Generation Real-Time Rendering with DXR, Vulkan, and OptiX*. Ed. by Adam Marrs, Peter Shirley, and Ingo Wald. Berkeley, CA: Apress, 2021, pp. 545–550. ISBN: 978-1-4842-7185-8. DOI: `10.1007/978-1-4842-7185-8_34`. URL: `https://doi.org/10.1007/978-1-4842-7185-8_34`.

[WB19] Carsten Wächter and Nikolaus Binder. “A Fast and Robust Method for Avoiding Self-Intersection: High-Quality and Real-Time Rendering with DXR and Other APIs”. In: Feb. 2019, pp. 77–85. ISBN: 978-1-4842-4426-5. DOI: `10.1007/978-1-4842-4427-2_6`.

[Wel62] B. P. Welford. “Note on a Method for Calculating Corrected Sums of Squares and Products”. In: *Technometrics* 4.3 (1962), pp. 419–420. DOI: `10.1080/00401706.1962.10490022`. eprint: `https://www.tandfonline.com/doi/pdf/10.1080/00401706.1962.10490022`. URL: `https:`

//www.tandfonline.com/doi/abs/10.1080/00401706.1962.10490022.

[WH06] Ingo Wald and Vlastimil Havran. "On building fast kd-Trees for Ray Tracing, and on doing that in O(N log N)". In: *2006 IEEE Symposium on Interactive Ray Tracing*. 2006, pp. 61–69. DOI: 10.1109/RT.2006.280216.

[WP19] Ingo Wald and Steven G. Parker. "RTX Accelerated Ray Tracing with OptiX". In: *ACM SIGGRAPH 2019 Courses*. SIGGRAPH '19. Los Angeles, California: Association for Computing Machinery, 2019. ISBN: 9781450363075. DOI: 10.1145/3305366.3340297. URL: https://doi.org/10.1145/3305366.3340297.

A CERN hardware

CERN offers strict access only on demand to its HPC facilities as most applications are focused on data processing. The general computer centre is a HTC facility (High Throughput Computing). One is the batch service dedicated to parallel computing on one node, the other is the HPC cluster. Both services allow MPI jobs to be run.

Batch Service

For applications running well on one node, but benefit from many cores the intended computing service is the batch system HTCondor. Multi-core jobs with up to 32 cores (with special requirement also 48 cores) are able to run on a node.

HPC/MPI Cluster

Access to the MPI Cluster is only allowed to specific user groups (mainly accelerator and technology sector). The HPC cluster is using SLURM as a workload manager and is accesible via lxplus or ssh.[1] Mvapich2 or OpenMPI are offered with SLURM. The default comppiler is gcc 4 and at least gcc 6 is also supported. Access can be granted via the e-group `hpc-plasma-users`.[2]

The homogeneous computing nodes for general access consist of the following hardware:[3]

- CPU: 2x Intel(R) Xeon(R) CPU E5-2650 v2 (16 physical cores, 32 hyper-threaded)
- Memory: 128GB DDR3 1600Mhz (8x16GiB M393B2G70QH0-YK0 DIMMs)
- Network: Chelsio T520-LL-CR (low-latency 10Gbit ethernet)
- Storage:

[1]`https://cern.service-now.com/service-portal/article.do?n=KB0004541`
[2]Accessed on 05/12/2022: `https://cern.service-now.com/service-portal/article.do?n=KB0004975`
[3]Accessed on 05/12/2022: `http://batchdocs.web.cern.ch/batchdocs/linuxhpc/resources.html`

  - CephFS for /hpscratch
  - 2TB HGST HUS724020AL for local scratch.

- about 100 nodes[4]

[4]Accessed on 05/12/2022: `https://indico.cern.ch/event/771821/contributions/3207080/attachments/1754919/2844956/HPC-CERN-2018.pdf`