



FEDERAL UNIVERSITY OF SANTA CATARINA
TECHNOLOGY CENTER
AUTOMATION AND SYSTEMS ENGINEERING DEPARTMENT
UNDERGRADUATE COURSE IN CONTROL AND AUTOMATION ENGINEERING

Arthur de Oliveira Machado

**Benchmarking Open-Source Libraries for 3D Object Comparison in Machining
Part Quality Control and Assessment**

Aachen, Germany
2025

Arthur de Oliveira Machado

**Benchmarking Open-Source Libraries for 3D Object Comparison in Machining
Part Quality Control and Assessment**

Final report of the subject DAS5511 (Course Final Project) as a Concluding Dissertation of the Undergraduate Course in Control and Automation Engineering of the Federal University of Santa Catarina.
Supervisor: Prof. Carlos Barros Montez
Co-supervisor: Sven Schiller

Aachen, Germany
2025

Ficha catalográfica gerada por meio de sistema automatizado gerenciado pela BU/UFSC.
Dados inseridos pelo próprio autor.

Machado, Arthur de Oliveira
Benchmarking Open-Source Libraries for 3D Object
Comparison in Machining Part Quality Control and
Assessment / Arthur de Oliveira Machado ; orientador,
Carlos Barros Montez, coorientador, Sven Schiller, 2025.
100 p.

Trabalho de Conclusão de Curso (graduação) -
Universidade Federal de Santa Catarina, Centro Tecnológico,
Graduação em Engenharia de Controle e Automação,
Florianópolis, 2025.

Inclui referências.

1. Engenharia de Controle e Automação. 2. Benchmarking.
3. 3D Object Comparison. 4. Quality Control. I. Montez,
Carlos Barros. II. Schiller, Sven. III. Universidade
Federal de Santa Catarina. Graduação em Engenharia de
Controle e Automação. IV. Título.

Arthur de Oliveira Machado

**Benchmarking Open-Source Libraries for 3D Object Comparison in Machining
Part Quality Control and Assessment**

This dissertation was evaluated in the context of the subject DAS5511 (Course Final Project) and approved in its final form by the Undergraduate Course in Control and Automation Engineering

Florianópolis, July 29-th, 2025.

Prof. Marcelo De Lellis Costa de Oliveira, Dr.
Course Coordinator

Examining Board:

Prof. Carlos Barros Montez, Dr.
Advisor
UFSC/CTC/EAS

Sven Schiller, M.Sc.
Supervisor
gemineers GmbH

Prof. Rodrigo Lange, Dr.
Evaluator
IFRS

Prof. Eduardo Camponogara, Dr.
Board President
UFSC/CTC/EAS

This work is dedicated to my classmates and my dear
parents.

ACKNOWLEDGEMENTS

I would like to begin by thanking my parents, Eduardo and Adriana, who have always supported me in any path I chose to follow and I know they will continue to support me in the journeys that lie ahead. I certainly would not be here without their unwavering support and all the love they have given me. I hope I can give back everything they have provided me throughout my life and make them proud.

I am also grateful to the friends I made along this academic journey, who made everything much easier and more enjoyable. It truly could not have been any other way. I would especially like to thank the friends I made in the PNA group, these are friendships I hope to carry with me for life, as they were there for me in the moments I needed it most, offering companionship, advice, and laughter.

I would also like to thank my colleagues with whom I had the honor of working alongside during this last year while writing my thesis: Maurício, Bruno, Pedro, Victor, Ícaro, Maria, Lucas, Giovanna, Milena, Rhanna, Paulo, Fernando, Piccoli, Maurici, Allan, Laranjinha, Luiz and Victória. You made the daily work routine something truly enjoyable and something I looked forward to every day. I am grateful to you not only for the work we did together but also for the experiences and trips we shared, which certainly helped us relieve the stress from work and life in general. I hope we can continue to nurture the friendship we have built over this time.

Finally, I would like to thank my supervisor at the company, Sven, for his leadership and Tommy for always providing ideas and guidance to solve any problem I encountered. I also thank all the professors at UFSC who made this project possible.

*"No one knows what the future holds, that's why its potential is infinite."
(Rintarou, 2011)*

DISCLAIMER

Aachen, July 29-th, 2025.

As representative of the gemineers GmbH in which the present work was carried out, I declare this document to be exempt from any confidential or sensitive content regarding intellectual property, that may keep it from being published by the Federal University of Santa Catarina (UFSC) to the general public, including its online availability in the Institutional Repository of the University Library (BU). Furthermore, I attest knowledge of the obligation by the author, as a student of UFSC, to deposit this document in the said Institutional Repository, for being it a Final Program Dissertation (*“Trabalho de Conclusão de Curso”*), in accordance with the *Resolução Normativa n° 126/2019/CUn*.



Sven Schiller
gemineers GmbH

ABSTRACT

This work addresses the critical need for performance optimization in quality control platforms for machining processes, a challenge faced by gemineers GmbH, whose proprietary software exhibits performance bottlenecks when handling complex geometries, such as blisks. The closed-source nature of the current solution prevents direct optimizations, motivating the search for more efficient and flexible open-source alternatives. The main objective of this research is to perform a benchmarking of open-source libraries to develop a high-performance alternative for 3D Object comparison in the quality control of machined parts. The project's methodology combines the agile Scrum framework for development and a structured process Benchmarking approach for the systematic evaluation of alternatives. Solutions were investigated and implemented using a set of Python libraries, including Open3D, PyVista and Vedo, focusing on techniques such as ray casting and nearest neighbor search for geometric deviation calculation. The results demonstrate that, although the quality of the input mesh is a critical factor for accuracy, a hybrid approach that combines the fast file loading of PyVista with the highly efficient distance calculation by ray casting from Open3D significantly outperformed both the proprietary software and the other tested configurations, with a notable reduction in execution time and memory consumption. The impact for the company is the validation of a high-performance open-source solution, which allows for the replacement of the proprietary tool, increasing the efficiency of its digital twin-based assessment platform and offering greater architectural flexibility for future developments.

Keywords: Benchmarking. 3D Object Comparison. Quality Control.

RESUMO

Este trabalho aborda a necessidade crítica de otimização de desempenho em plataformas de controle de qualidade para processos de usinagem, um desafio enfrentado pela gemineers GmbH, cujo software proprietário apresenta gargalos de performance ao lidar com geometrias complexas, como os blisks. A natureza fechada da solução atual impede otimizações diretas, motivando a busca por alternativas de código aberto mais eficientes e flexíveis. O objetivo principal desta pesquisa é realizar um Benchmarking de bibliotecas open-source para desenvolver uma alternativa de alto desempenho para a comparação de objetos 3D no controle de qualidade de peças usinadas. A metodologia do projeto combina o framework ágil Scrum para o desenvolvimento e uma abordagem estruturada de Benchmarking de processos para a avaliação sistemática das alternativas. Foram investigadas e implementadas soluções utilizando um conjunto de bibliotecas Python, incluindo Open3D, PyVista e Vedo, com foco em técnicas como ray casting e busca de vizinhos próximos para o cálculo de desvio geométrico. Os resultados demonstram que, embora a qualidade da malha de entrada seja um fator crítico para a precisão, uma abordagem híbrida que combina o carregamento rápido de arquivos do PyVista com o cálculo de distância por ray casting altamente eficiente do Open3D oferece uma alternativa que supera significativamente tanto o software proprietário quanto as outras configurações testadas, com notável redução no tempo de execução e no consumo de memória. O impacto para a empresa é a validação de uma solução open-source de alto desempenho, que possibilita a substituição da ferramenta proprietária, aumentando a eficiência de sua plataforma de avaliação com gêmeos digitais e oferecendo maior flexibilidade arquitetônica para futuros desenvolvimentos.

Palavras-chave: Benchmarking. Comparação de objetos 3D. Controle de qualidade.

LIST OF FIGURES

Figure 1 – Example of a Blisk.	17
Figure 2 – Benchmarking matrix for recommended combinations of types of benchmarking.	22
Figure 3 – The benchmarking wheel.	23
Figure 4 – gemineers GmbH logo.	25
Figure 5 – VEF Graph to describe an example geometry.	29
Figure 6 – Triangulation of a surface.	29
Figure 7 – Wireframe model of a gearbox.	30
Figure 8 – Parametric modeling in a CAD program.	30
Figure 9 – Example structure of a binary STL file.	31
Figure 10 – Example of a cube definition in a binary PLY file.	32
Figure 11 – Example glTF file of a Mesh.	34
Figure 12 – Illustration of raycasting.	35
Figure 13 – An example of a BVH.	36
Figure 14 – An example of a level set model with the signed distance function.	37
Figure 15 – Diagram illustrating different execution models: single processor single thread, single processor multi-thread, multi-processor single thread and multi-processor multi-thread, along with key memory components.	40
Figure 16 – Finish parts for the tested meshes.	41
Figure 17 – Digital Twins generated by proprietary software with different Deviations for Test Brick.	45
Figure 18 – Digital Twins generated by proprietary software with different Deviations for PSM Demo.	46
Figure 19 – Digital Twins generated by proprietary software with different Deviations for Jet Bracket.	47
Figure 20 – Digital Twins generated by proprietary software with different Deviations for Demo Blisk.	48
Figure 21 – Proposed approach for evaluating the performance of test objects.	52
Figure 22 – Mean loading times of different file formats.	55
Figure 23 – Proposed approach to evaluate the calculation of deviation for libraries.	58
Figure 24 – Center of each triangle in a Teapot mesh.	60
Figure 25 – Digital Twins generated by Open3D with different Deviations for Test Brick workpiece.	63
Figure 26 – Wireframe model of the Test Brick Finish Part.	64
Figure 27 – Wireframe model of the PSM Demo. Finish Part.	64
Figure 28 – Digital Twins generated by Open3D with different Deviations for PSM Demo. workpiece.	65

Figure 29 – Direct Comparison between triangles and their deviation values.	65
Figure 30 – Proprietary software test with 0.15 internal precision resolution.	66
Figure 31 – Digital Twins generated by Open3D with different Deviations for the rest of the workpieces.	67
Figure 32 – Digital Twins generated by Pyvista with different Deviations for Test Brick workpiece.	70
Figure 33 – Digital Twins generated by Pyvista with different Deviations.	71
Figure 34 – Digital Twins generated by Vedo with different Deviations for Test Brick workpiece.	73
Figure 35 – Digital Twins generated by Vedo with different Deviations.	74
Figure 36 – Digital Twins generated by Vedo with different Deviations for Demo Blisk workpiece.	75
Figure 37 – "Shadow-like" artifact in the demoblisk digital twin generated by Vedo.	75
Figure 38 – Digital Twin generated by new normal calculation process in Vedo.	76
Figure 39 – Open3D Memory Usage segmented by sections.	77
Figure 40 – Monitoring Metrics comparison between Open3D and Transformation methods.	79
Figure 41 – Digital twins produced by frontend implementation with 0.03 mm de- viation.	81
Figure 42 – MeshLab Interface.	86
Figure 43 – Color Mapping Result for PSM Demo.	87
Figure 44 – CloudCompare Interface.	91
Figure 45 – Color mapping for PSM Demo. at 0.03 mm.	92

LIST OF FRAMES

Frame 1 – Types of benchmarking and their definitions.	21
Frame 2 – Types of benchmarking and their definitions.	22
Frame 3 – Comparison of Open-Source Libraries.	50

LIST OF TABLES

Table 1 – Tests with Internal Precision Setting of 0.7.	44
Table 2 – Tests with Internal Precision Setting of 0.5.	44
Table 3 – Tests with Internal Precision Setting of 0.3.	44
Table 4 – Open3D metrics.	62
Table 5 – Pyvista metrics.	69
Table 6 – Vedo metrics.	72
Table 7 – Transforming metrics (Vedo - Open3D).	78
Table 8 – Transforming metrics (Pyvista - Open3D).	78
Table 9 – VM Open3D metrics.	83
Table 10 – VM Pyvista metrics.	83
Table 11 – VM Vedo metrics.	84
Table 12 – VM Transforming metrics (Vedo - Open3D).	84
Table 13 – VM Transforming metrics (Pyvista - Open3D).	84
Table 14 – Signed distances metrics for PSM Demo. (MeshLab).	87
Table 15 – Signed distances metrics for Alpha workpiece (MeshLab).	89
Table 16 – Signed distances metrics for new Alpha workpiece manifold (MeshLab).	90
Table 17 – Signed distances metrics for PSM Demo. (CloudCompare).	92
Table 18 – Signed distances metrics for Alpha workpiece. (CloudCompare).	93

LIST OF ABBREVIATIONS AND ACRONYMS

BVH	Bounding-Volume Hierarchy
CAD	Computer-Aided Design
GLTF	GL Transmission Format
PLY	Polygon File Format
STL	Standard Tessellation Language
VTK	Visualization Toolkit

CONTENTS

1	INTRODUCTION	17
1.1	MOTIVATION	17
1.2	OBJECTIVES	18
1.3	DOCUMENT STRUCTURE	19
2	METHODOLOGY	20
2.1	THE SCRUM FRAMEWORK	20
2.2	RESEARCH APPROACH	21
3	THE COMPANY AND ITS SOFTWARE	25
3.1	CURRENT SOFTWARE	25
3.1.1	Data Ground	25
3.1.2	Data Processing	26
3.1.3	Third party	26
3.1.4	Interfaces	26
3.1.4.1	REST API	26
3.1.4.2	Front end	26
4	THEORETICAL BACKGROUND	28
4.1	GEOMETRIC MODELING	28
4.2	RAY CASTING	35
4.3	BOUNDING VOLUME HIERARCHY	36
4.4	SIGNED DISTANCE FUNCTION	37
4.5	PROFILING AND PERFORMANCE ANALYSIS	38
4.5.1	Overhead	38
4.5.2	Sampling	38
4.6	CONCURRENT AND PARALLEL COMPUTING	39
5	TESTING AND BENCHMARKING	41
5.1	TEST OBJECTS	41
5.2	CURRENT CODE ANALYSIS	43
5.3	INVESTIGATION OF OPEN-SOURCE LIBRARIES	49
5.4	TEST ENVIRONMENT SETUP	52
5.5	FRONTEND PARSING	55
6	IMPLEMENTATION OF ALTERNATIVES	57
6.1	ACTIVITY DIAGRAM	57
6.2	OPEN 3D	62
6.3	PYVISTA	68
6.4	VEDO	72
6.5	TRANSFORMING	77
6.6	TRIMESH	79

6.7	FRONTEND	80
6.7.1	BVH	81
6.8	COMPARATIVE ANALYSIS	82
6.9	BENCHMARKING WITH NON-COMPETITOR COMPANIES	86
7	CONCLUSION AND FUTURE WORK	94
	References	96
	ANNEX A – CODES USED AS EXAMPLES IN THE PROJECT . .	100

1 INTRODUCTION

This chapter is an introduction to the project, presenting its context, motivation and objectives, and outlines the problem that drives the study. Additionally, it provides an overview of the document's structure, detailing the content and purpose of each chapter to guide the reader through the research.

1.1 MOTIVATION

In today's competitive industrial landscape, precision and efficiency in machining processes are critical for maintaining product quality and customer satisfaction. Companies face increasing pressure to meet these demands while minimizing costs and improving operational efficiency. With that in mind, gemineers GmbH specializes in providing a quality assessment platform for machining processes using digital twins.

A prime example of the challenges faced in machining processes can be seen in the production of blisks (blade-integrated disks). These components are vital in industries such as aerospace and energy due to their lightweight, high-strength properties and the precision required in their manufacturing (Dilba, 2024). Blisks feature complex geometries, as seen in Figure 1 and demand extremely tight tolerances, requiring high-fidelity 3D representations for effective quality assessment. However, such complexity significantly increases computational loads, particularly in systems where execution time, memory usage and CPU performance are critical. The proprietary software currently used for quality assessment struggles to handle these demands efficiently. Its closed nature prevents code-level optimizations, further exacerbating bottlenecks as computational requirements grow.

Figure 1 – Example of a Blisk.



Source: (ZEISS, 2024)

Despite their widespread use, proprietary solutions often face significant challenges. Their closed nature limits the ability to adapt to specific workflows, making them less scalable for industries with unique or evolving requirements (Prasad; Reddy, 2013). Moreover, addressing performance bottlenecks, such as optimizing execution time, memory usage and CPU performance, can be difficult, as users lack the flexibility to modify or customize the software's internal mechanisms.

In response to these limitations, many organizations have turned to open-source software as an alternative to also reduce expenses and remain competitive in the market. Open-source software allows unrestricted access to its source code, permitting redistribution without charge, enabling modifications and creating derivative works that can also be redistributed under the same terms as the original software (O'Reilly, 1999). These platforms enable rapid integration of cutting-edge algorithms, making it easier to address performance issues and adapt to evolving needs (Elpern; Dascalu, 2009). By offering flexibility and transparency, open-source solutions allow developers to optimize computational performance, reducing execution time and improving resource utilization. This is particularly valuable in high-precision industries like machining, where even minor inefficiencies can significantly impact productivity and costs. Considering this, many open-source software products have become powerful rivals to proprietary software.

1.2 OBJECTIVES

The primary objective of this research is to benchmark open-source alternatives against the proprietary library currently in use, assessing their ability to meet or exceed the performance and accuracy requirements. Additionally, comparisons will be made with solutions developed by non-competitor companies within the same industry or technological domain to provide a broader perspective on performance.

Based on this stipulated objective, specific tasks have been defined in order to achieve a functional alternative that is well suited to the gemineers' project as a whole:

- Gather current solution metrics and analyze the bottlenecks;
- Research open-source libraries that have the same functionalities as the current one and if possible more;
- Verify the possibility of diversing the current workflow by handling different file formats;
- Implement a new alternative with the best libraries and compare them;
- Propose if the new alternative can be done inside of the frontend and eliminate the idea of a separated service;

- Benchmarking of the implemented alternatives with non-competitor companies within the same industry or technological area;
- Evaluate the results obtained.

1.3 DOCUMENT STRUCTURE

This document is divided into nine chapters to detail the development of the project. Chapter 2 explains the different approaches and steps for the project's planning and execution.

In Chapter 3, the company's context is explored alongside an introduction to the gemineers software.

Chapter 4 establishes the theoretical foundation for the research, covering key concepts of geometry processing and detailing the related technologies, as well as the fundamentals and state-of-the-art advances of the main topics of the development.

Chapter 5 presents the testing and benchmarking process used to evaluate the different software solutions. It details the testing methodology, the performance metrics (execution time, memory and CPU usage), the research of the open-source libraries and the standard test objects used for the evaluation. The chapter also includes a baseline analysis of the existing proprietary software, providing a benchmark against which the open-source alternatives are compared.

The details of the implementation of the chosen open-source alternatives are described in Chapter 6. It describes how these solutions replicate or improve upon the functionality of the proprietary software and discusses the challenges faced during integration and the strategies used to address them.

Finally, Chapter 7 summarizes the findings of the research, reflecting on the success of the proposed solutions in meeting the objectives. It also makes suggestions for future research and next steps in the context of the project.

2 METHODOLOGY

This chapter describes the methodology used in the project to achieve the objective stated in the previous chapter, alongside the research approach used to perform the specific tasks that were aligned.

2.1 THE SCRUM FRAMEWORK

Scrum is an iterative, incremental framework for projects and product or application development (Sutherland, 2025). The workflow is organized into fixed-length cycles known as Sprints, which typically last less than one month. During each Sprint, specific items are selected from a prioritized list of potential product improvements, referred to as the Product Backlog. The development team commits to completing these selected items within the Sprint, aiming to enhance the product's value by the end of the cycle. According to (Sutherland, 2025), the Scrum framework is based on a Scrum Team that consists of three roles:

- **Product Owner:** responsible for gathering input on what the product should be and translating it into a clear product vision or a Product Backlog, with the goal of maximizing the product's value.
- **The Team:** composed of individuals tasked with developing the product as envisioned by the Product Owner.
- **Scrum Master:** a facilitator who ensures the success of the Scrum Team by applying the Scrum framework. This role involves removing organizational impediments, facilitating meetings and helping all team members understand and apply Scrum theory and practices effectively.

The key events that enable the development team to deliver value through this iterative process are as follows (Sutherland, 2025):

- **Sprint Planning:** this event marks the start of each Sprint and involves the entire Scrum Team. During the session, the team reviews the Product Backlog, discusses the goals and context of the items and selects the tasks they commit to completing by the end of the Sprint.
- **Daily Standup:** a brief, 15-minute meeting held each workday, where team members provide updates on their progress, discuss challenges or impediments and coordinate their next steps.
- **Sprint Review:** Conducted at the end of the Sprint, this meeting provides an opportunity to present completed work to stakeholders. It allows the team to inspect the Sprint's outcome and gather feedback to inform future planning.

- **Sprint Retrospective:** held after the Sprint Review, this event focuses on evaluating the team's performance. The team reflects on what worked well, identifies areas for improvement and agrees on actionable changes to enhance future Sprints. This involves reviewing interactions, processes and tools used during the Sprint.

2.2 RESEARCH APPROACH

For this project, it is essential to establish two key concepts: The first is the concept of benchmarking from a computer science and technological perspective and the second concept is benchmarking in the context of business processes.

The foremost refers to the process of running standardized tests to evaluate the relative performance of systems, algorithms, or products (Dunham; West, 2011; Sim; Easterbrook; Holt, 2003). This concept will be applied to analyze the newly identified open-source libraries and the developed implementation in comparison to the proprietary software.

While the latest involves the continuous measurement and comparison of an organization's processes against comparable practices in leading organizations to gather insights that drive improvement (Andersen; Pettersen, 1995). This concept will be used to compare the results of the implementations with those of non-competing companies, ensuring that the solutions align with industry best practices and standards.

Three types of benchmarking can be defined, depending on what is being compared in Frame 1:

Frame 1 – Types of benchmarking and their definitions.

Benchmarking Type	Definition
Performance benchmarking	Comparison of performance measures, for the purpose of determining how good one's own company is compared to others.
Process benchmarking	Comparison of method and practices for performing business processes, for the purpose of learning from the best to improve one's own processes.
Strategic benchmarking	Comparison of the strategic choices and dispositions made by other companies, for the purpose of collecting information to improve one's own strategic planning and positioning.

Source: Adapted from (Andersen; Pettersen, 1995)

Four types of benchmarking can be defined in Frame 2, depending on whom one compare against:

Frame 2 – Types of benchmarking and their definitions.

Benchmarking Type	Definition
Internal	Comparison between departments, units, subsidiaries or countries within the same company or organisation.
Competitive	Direct comparison of own performance/results against the best real competitors.
Functional	Comparison of processes or functions against non-competitor companies within the same industry or technological area.
Generic	Comparison between own processes against the best processes around, regardless of industry.

Source: Adapted from (Andersen; Pettersen, 1995)

The type of benchmarking most suitable for the work presented in this project is Process Benchmarking. This approach focuses on improving gemineers' machining part quality control and assessment service by evaluating whether the results align with industry standards and best practices.

To identify the most appropriate benchmarking strategy and determine the relevant comparisons, Figure 2 provides a structured framework for selecting the optimal type of benchmarking and identifying the entities to benchmark against.

Figure 2 – Benchmarking matrix for recommended combinations of types of benchmarking.

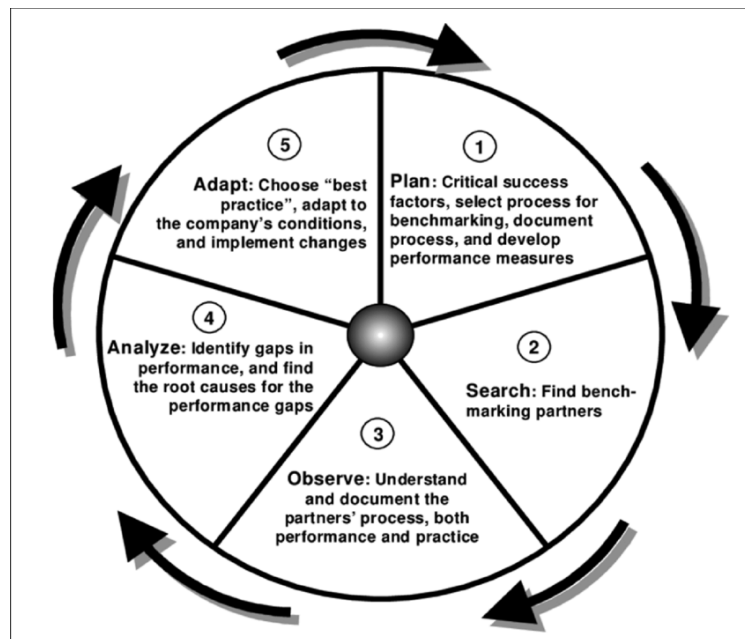
	Internal benchmarking	Competitor benchmarking	Functional benchmarking	Generic benchmarking
Performance Benchmarking	∇	Φ	∇	θ
Process Benchmarking	∇	θ	Φ	Φ
Strategic Benchmarking	θ	Φ	θ	θ
Relevance/Value		High Φ	Medium ∇	Low θ

Source: (Bhutta; Huq, 1999)

Figure 2 supports the conclusion outlined in the Section 1.2: when applying Process Benchmarking, it is beneficial to also incorporate Functional or Generic Benchmarking. This combined approach provides a broader perspective on the company's strengths and weaknesses, helping to identify areas for improvement and actionable steps to enhance performance.

At this stage, defining a clear course of action for the planned activities is necessary. To achieve this, the benchmarking wheel in Figure 3 will be used.

Figure 3 – The benchmarking wheel.



Source: Adapted from (Camp, 1989)

The first part of the wheel is responsible for the planning and it is said as the most important part of all phases. The activities that are performed in this phase can be described as:

1. Selection of the process to be benchmarked based on the strategy of the company;
2. Select the Benchmarking team;
3. Understand and document the process to be benchmarked;
4. Establish performance measures for the process (quality, time, cost, etc.).

Most of these steps have already been completed by gemineers' Product Owner. The understanding and establishment of performance measurements were collaboratively carried out with the Benchmarking team, which includes the author of this project.

In accordance with the benchmarking wheel, the Search and Observe steps can be combined into a single integrated step. In this step, the first task is to design a list of criteria that an ideal benchmarking process should meet. Next, the search is conducted to identify candidates that align with these criteria. Once identified, their processes and performance are observed in detail. The information gathered during this phase is then documented for subsequent analysis.

In the Analyze step, we identify gaps in performance between our processes and those of the benchmarking partners. This involves uncovering the root causes of these gaps and examining the methods and practices that enable the partners to achieve their high-performance levels. For this project, this step is used to compare the results of the developed methods with the existing ones, evaluating whether the improvements align with the partners' standards.

The final stage of the wheel is Adapt, where the insights from the analysis are tailored to the organization's specific conditions and implemented within the company.

The integration of the Scrum framework and the benchmarking methodology provides a structured and efficient approach to enhancing gemineers' machining part quality control and assessment service. By combining Process Benchmarking with Functional and Generic comparisons and leveraging the systematic steps of the benchmarking wheel, this project establishes a robust foundation for identifying performance gaps, adopting best practices and driving continuous improvement aligned with industry standards.

3 THE COMPANY AND ITS SOFTWARE

The gemineers GmbH is a spin-off startup from the Fraunhofer Institute for Production Technology (Fraunhofer IPT), officially founded in July 2021. Based in Aachen, Germany, the company emerged from the High-Performance Cutting department, renowned for its innovations in simultaneous multi-axis milling and advanced machining solutions for turbomachinery and aerospace applications (FRAUNHOFER, 2025).

The company specializes in software development for manufacturing analysis, with a focus on accelerating the digitalization of machining processes. Its flagship product leverages digital twin technology to enable clients to monitor machining operations in real-time. By collecting data directly from machine tools, the software provides a comprehensive digital record of machining products and processes. This technology simplifies quality assessment, enhances process evaluation and facilitates data-driven decision-making.

Figure 4 – gemineers GmbH logo.



Source: Company's archive

3.1 CURRENT SOFTWARE

The gemineers software is built on modular architecture, where the application is composed of multiple autonomous services that work collaboratively. Each service is loosely coupled, can be deployed independently, and is designed to perform a specific task. Communication between these services occurs via simple interfaces, such as communication and messaging protocols. This approach offers significant flexibility and easier maintainability, enabling the delivery of a scalable and reliable product to the company's customers. To achieve this, the software is divided into four key areas, each containing specific services. The following subsections provide a detailed explanation of these areas and the services they encompass.

3.1.1 Data Ground

The Data Ground module serves as the essential starting point in the software's data processing chain. It is responsible for acquiring real-time data directly from machine tools during machining operations. By capturing crucial parameters such as tool

positions, feed rates, spindle speeds and cutting forces, it ensures the accurate and comprehensive collection of machine information. A key responsibility of this module is to transform raw, heterogeneous data into a standardized and machine-readable format. This standardization process harmonizes inputs from various sensors and machine tools, creating a unified data stream that ensures compatibility with downstream services. As a result, the entire system can efficiently process, analyze and report critical machining insights across different types of equipment and operational environments.

3.1.2 Data Processing

The Data Processing module plays a critical role in refining and analyzing the standardized data acquired by the Data Ground service. In order to perform the necessary calculations, simulations and predictions required for developing an accurate digital twin of the final workpiece, this area is further subdivided into other specialized services, each focused on distinct tasks within the refinement and analytical processes.

3.1.3 Third party

This area serves as the central intelligence of the entire application, orchestrating communication between various services.

3.1.4 Interfaces

The Interfaces Module provides an intuitive and interactive platform for presenting the data acquired, processed and generated by gemineers services, as well as information directly from the machine. Its user-friendly design aims to streamline decision-making by delivering clear and actionable insights. The two primary microservices of this area are:

3.1.4.1 REST API

The REST API serves as the backbone for data integration, facilitating seamless communication between services and external systems. It manages access to the software's database while handling user authentication and authorization to ensure secure interactions. This service plays a critical role in orchestrating data flow, maintaining system integrity and supporting efficient service integration across the platform.

3.1.4.2 Front end

This area is dedicated to delivering an intuitive and interactive web interface for visualizing and managing both acquired and simulated data. Powered by a modern web framework, it enables users to view toolpath details, inspect deviations and monitor tool engagement in real time. Additionally, it provides real-time feedback on metrics

such as tool speed, engagement zones and deviation levels, allowing users to quickly identify anomalies, such as excessive tool wear or improper material removal, before they impact the final product quality.

4 THEORETICAL BACKGROUND

4.1 GEOMETRIC MODELING

Geometric modeling focuses on the mathematical representation of shapes and real-world objects, playing a vital role in Computer-Aided Design (CAD) and computer graphics (Nguyen; Tran; Maleszka, 2021). It integrates concepts from geometry, trigonometry and differential geometry to create accurate digital models used in design, simulation and visualization (Penev, 2013). As a technique for describing objects in multidimensional space (Borrmann *et al.*, 2018), geometric modeling is essential in fields like architecture and engineering.

The following subsections introduce the four main types of geometric modeling, each with its distinct characteristics and applications.

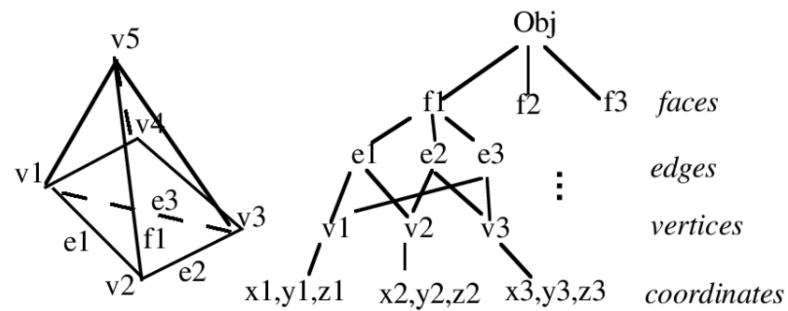
Solid modeling

There are two main approaches to modeling 3D bodies: explicit modeling and implicit modeling (Borrmann *et al.*, 2018). Explicit modeling represents a volume using geometric elements such as vertices, edges and surfaces, allowing precise control over the shape. In contrast, implicit modeling defines a volumetric body indirectly through mathematical functions, such as level sets or signed distance functions (SDFs). This method is particularly useful for complex organic shapes and smooth transitions, as seen in volumetric modeling and constructive solid geometry.

Within the scope of this project, only explicit modeling will be considered and utilized. This approach typically involves two common methods for creating an object:

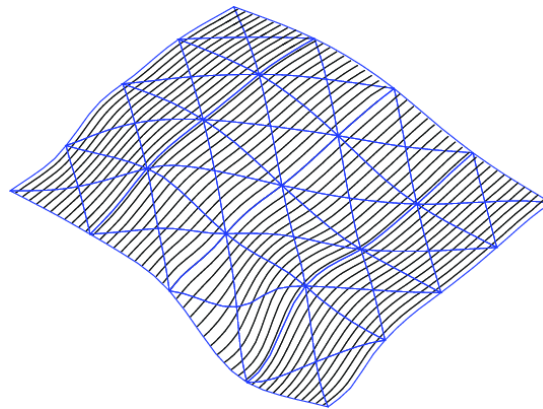
- **Boundary representation methods:** is a widely used method for describing 3D bodies in computing. It defines a hierarchy of elements: Body, Face, Edge and Vertex. Each element is described by the lower-level elements, such as a body by its faces, a face by its edges and an edge by its start and end vertices (Borrmann *et al.*, 2018). This hierarchy establishes the topology of the body and can be represented as a vertex-edge-face (VEF) graph (Fig. 5).
- **Triangulated surface modeling:** A simplified version of boundary representation is describing a body's surface as a triangle mesh (Borrmann *et al.*, 2018). Although curved surfaces cannot be described exactly, they can be approximated by using a finer mesh, which increases accuracy. The 3D object in this method is represented as a set of faces, as shown in Figure 6. It consists of a wireframe with gaps between edges filled using a specific method.

Figure 5 – VEF Graph to describe an example geometry.



Source: (De-cambray; Delpy; Zeitouni, 1995)

Figure 6 – Triangulation of a surface.

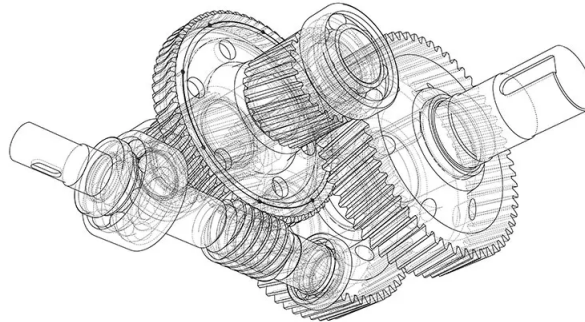


Source: (Zhang *et al.*, 2019)

Wireframe modeling

Wireframe modeling, while no longer the primary technique for 3D shape representation, is still useful for quickly displaying simple objects, especially for projections (Woodwark, 1986). It shows only edges, as seen in Figure 7, making it simple and efficient but lacking detail, which can hinder visualization.

Figure 7 – Wireframe model of a gearbox.

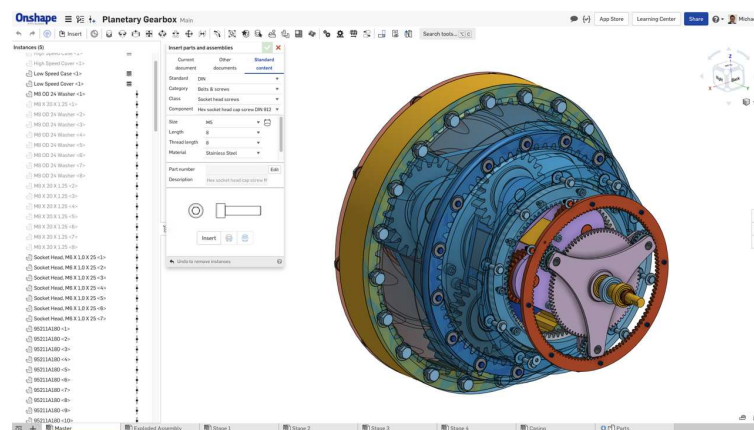


Source: (Systèmes, 2025)

Parametric modeling

Parametric modeling allows for automatic adjustments to model geometry when dimensions are changed (Fu, 2018). It uses programming scripts to define the model's shape and dimensions, visualized in 3D drafting programs. Common in the building sector, parametric modeling relies on dependencies and constraints, making it flexible and adaptable to changes. Parameters like height, width and position are defined and relationships between them are controlled through equations, ensuring that changes, such as in story height, automatically update related elements like wall heights. This modeling is commonly used in programs like SolidWorks, FreeCAD and Fusion 360.

Figure 8 – Parametric modeling in a CAD program.



Source: (Cooke, 2025)

File formats

STL

The Standard Tessellation Language (STL) file format is widely used in 3D printing, rapid prototyping and computer-aided manufacturing (Iancu, C.; Iancu, D.; Stăncioiu, 2010). Originally developed by 3D Systems Inc., it has become a standard supported by many software applications. STL files represent only the surface geometry of a 3D object without color, texture, or other CAD attributes.

An STL file consists of an unstructured triangulated surface, where each triangle is defined by three vertices and a unit normal, following the right-hand rule in a three-dimensional Cartesian coordinate system (Iancu, C.; Iancu, D.; Stăncioiu, 2010). A greater number of triangles increases resolution, enabling more accurate representation of complex designs. However, while higher resolutions improve the quality of 3D models, they also significantly increase the STL file size. Excessively large files can make rendering and storage more challenging. Therefore, maintaining a balance between model quality and usability is essential.

The STL format supports both ASCII and binary representations, with binary being more common due to its smaller file size. A typical binary STL file begins with an 80-character header, which is generally ignored but should never start with "solid" to prevent software from misinterpreting it as an ASCII file. After the header, a 4-byte unsigned integer specifies the number of triangular facets in the file, followed by the data describing each triangle sequentially:

Figure 9 – Example structure of a binary STL file.

```

UINT8[80]      - Header
UINT32         - Number of triangles

for each triangle
  REAL32[3]    - Normal vector
  REAL32[3]    - Vertex 1
  REAL32[3]    - Vertex 2
  REAL32[3]    - Vertex 3
  UINT16       - Attribute byte count
end
```

Source: (Iancu, C.; Iancu, D.; Stăncioiu, 2010)

As shown in the Figure 9, the file provides a raw description of individual triangles without any connectivity information. Therefore, softwares that are used to visualize an STL file must interpret these triangles and reconstruct a cohesive 3D object.

PLY

The Polygon File Format (PLY) describes a 3D object as a collection of vertices, faces and other elements, each with associated properties such as color, normal direction, texture coordinates and transparency (Turk, 2025). Unlike STL, which only defines surface geometry, PLY supports richer data representation, making it widely used in 3D scanning, modeling and computational geometry.

A PLY file contains exactly one object, which may originate from sources like hand-digitized objects, polygonal models, range data, marching cubes (isosurface extraction), terrain data and radiosity models. The file typically includes a list of vertex coordinates (x, y, z) and face definitions referencing these vertices. However, it also allows the creation of new elements such as edges, material properties (ambient, diffuse and specular coefficients) and confidence values for range data.

The structure of a PLY file begins with an ASCII-formatted header defining the elements and their properties, followed by the actual data. The header always starts with a "magic number" line containing ply as seen in Figure 10, distinguishing it from other formats. The file can also be stored in ASCII or binary format. Programs that do not recognize certain properties or elements can either ignore or discard them, ensuring backward compatibility. This flexibility makes PLY a versatile format for 3D data storage and processing.

Figure 10 – Example of a cube definition in a binary PLY file.

```
ply
format ascii 1.0
comment author: Greg Turk
comment object: another cube
element vertex 8
property float x
property float y
property float z
property red uchar { start of vertex color }
property green uchar
property blue uchar
element face 7
property list uchar int vertex_indices { number of vertices for each face }
element edge 5 { five edges in object }
property int vertex1 { index to first vertex of edge }
property int vertex2 { index to second vertex }
property uchar red { start of edge color }
property uchar green
property uchar blue
end_header
0 0 0 255 0 0 { start of vertex list }
0 0 1 255 0 0
0 1 1 255 0 0
0 1 0 255 0 0
1 0 0 0 255
1 0 1 0 255
1 1 1 0 255
1 1 0 0 255
1 1 0 0 255
3 0 1 2 { start of face list, begin with a triangle }
3 0 2 3 { another triangle }
4 7 6 5 4 { now some quadrilaterals }
4 0 4 5 1
4 1 5 6 2
4 2 6 7 3
4 3 7 4 0
0 1 255 255 255 { start of edge list, begin with white edge }
1 2 255 255 255
2 3 255 255 255
3 0 255 255 255
2 0 0 0 0 { end with a single black line }
```

Source: (Turk, 2025)

GLTF

The GL Transmission Format (GLTF) is a widely used 3D file format designed by the Khronos Group for the efficient transmission and loading of 3D models across networks (Khronos, 2025). It has become a standard for real-time applications, including web-based 3D visualization, gaming and virtual reality, due to its optimized structure and lightweight nature.

At its core, GLTF uses a JSON file to define the structure and composition of a 3D scene. This JSON file organizes essential elements, including:

- **Scenes and Nodes:** Define the hierarchy and basic structure of the scene.
- **Meshes:** Contain the geometry of 3D objects, including vertices, indices and primitives.
- **Materials:** Specify how objects should be rendered, including shading, textures and lighting properties.
- **Skins:** Provide information for vertex skinning, crucial for animating characters and deformable objects.
- **Animations:** Describe how object properties change over time, supporting keyframe animation for transformations like translation, rotation and scaling.

In addition to the JSON structure, GLTF files may reference external binary data (GLB format includes this inline) for geometry and textures, ensuring fast loading and efficient rendering. Compared to formats like STL or PLY, GLTF is highly optimized for web applications, reducing file sizes while maintaining rich scene details, making it a preferred format for modern 3D workflows.

Figure 11 shows an example of such a file, generated using Open3D, which illustrates the JSON structure containing some of the key elements previously described, including the mesh, node and scene definitions.

Figure 11 – Example glTF file of a Mesh.

```
1 {  
2 >   "accessors": [...  
41 ],  
42   "asset": {  
43     "generator": "Open3D",  
44     "version": "2.0"  
45   },  
46 >   "bufferViews": [...  
61 ],  
62 >   "buffers": [...  
67 ],  
68   "meshes": [  
69     {  
70       "primitives": [  
71         {  
72           "attributes": {  
73             "NORMAL": 2,  
74             "POSITION": 1  
75           },  
76           "indices": 0,  
77           "mode": 4  
78         }  
79       ]  
80     }  
81   ],  
82 >   "nodes": [...  
86 ],  
87   "scene": 0,  
88 >   "scenes": [...  
94 ]  
95 }  
96 }
```

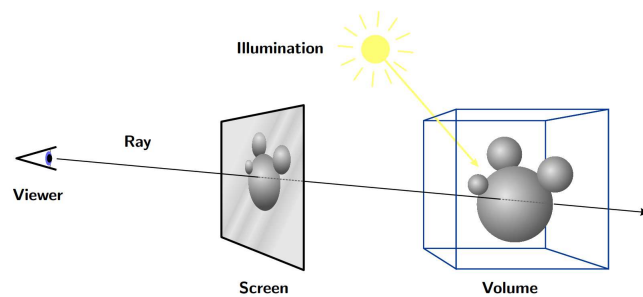
Source: Author's archive

4.2 RAY CASTING

Ray casting is a fundamental technique in computer graphics used for rendering and visibility determination. It is classified as an object-order rendering method, meaning that computations are performed based on the voxels in a volumetric model that intersect with each cast ray (Foley *et al.*, 1990).

The ray casting algorithm follows a structured process. A ray is emitted from a point on the virtual camera's projection plane and is traced until it intersects with an object in the virtual scene. This intersection determines which part of the object is visible to the observer. Figure 12 shows how this approach is made.

Figure 12 – Illustration of raycasting.



Source: (Ljung, 2006)

Mathematically, a ray is defined using a parametric equation:

$$r(t) = o + td \quad (1)$$

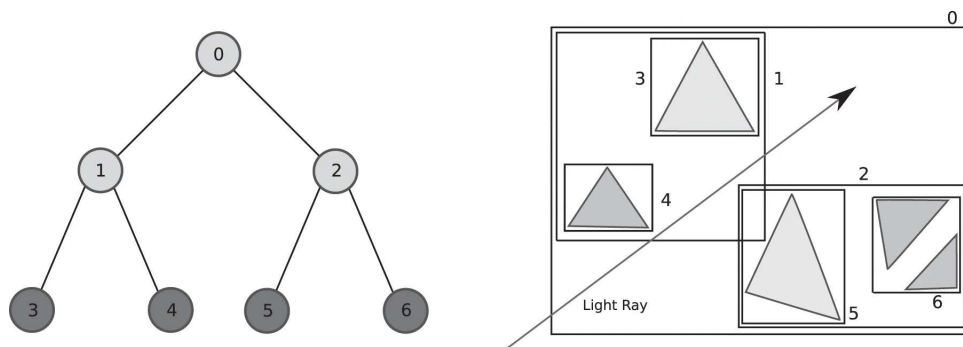
where o represents the camera position, d is the direction of the ray and t is a scalar that determines points along the ray's path. The angle of the ray's travel is influenced by the horizontal and vertical field-of-view (FoV), the ray's origin on the projection plane and the overall dimensions of the projection plane (Teschner, 2025). A zero-degree field-of-view results in an orthographic projection, meaning objects appear the same size regardless of depth.

Ray casting has numerous applications across different fields. In volumetric rendering, it is essential for medical imaging technologies like CT and MRI scans, where it helps generate 3D representations from 2D slices (Borzecki *et al.*, 2014). In 3D CAD/CAM modeling, it plays a crucial role in solid modeling and engineering design (Roth, 1982). The technique was also widely used in early video games, such as Wolfenstein 3D, to create efficient pseudo-3D environments without the computational overhead of full 3D rendering (Adobe, 2025). Additionally, ray casting is useful in physics simulations, particularly in collision detection, shadow calculations, and line-of-sight determination in artificial intelligence applications.

4.3 BOUNDING VOLUME HIERARCHY

One of the most prominent approaches to measuring distances between polygon meshes is to construct Bounding-Volume Hierarchies (BVHs) to approximate the shape of each mesh at iteratively finer levels (Larsen *et al.*, 1999). A Bounding-Volume Hierarchy (BVH) is a tree structure that organizes objects into a hierarchy of bounding volumes, such as Axis-Aligned Bounding Boxes (AABBs) or Oriented Bounding Boxes (OBBs), to accelerate spatial queries like ray tracing, ray casting and collision detection. By grouping objects into bounding volumes and employing heuristics like the Surface Area Heuristic (SAH) for optimization, BVHs reduce the number of intersection tests needed. During traversal, entire branches can be discarded if their bounding volumes do not intersect, significantly improving computational efficiency.

Figure 13 – An example of a BVH.



Source: (Doyle; Tuohy; Manzke, 2018)

Figure 13 illustrates the BVH process. On the left side, a BVH tree structure is depicted. Each node in the hierarchy represents a bounding volume that encloses multiple objects. The root node (0) contains all objects and as the tree is traversed downward, the objects are grouped into smaller bounding volumes (nodes 1 and 2), until reaching the leaf nodes (3, 4, 5 and 6), which contain the actual geometric primitives (e.g., triangles).

On the right side, the corresponding spatial representation of the BVH is shown. Each bounding volume encloses one or more triangles and the ray tracing process is demonstrated. Instead of checking for intersections with all triangles, the ray first tests the top-level bounding volume (0). If the ray does not intersect a bounding volume, all enclosed objects are skipped. In this case, the ray intersects bounding volume 1, so further checks are performed on its children (3 and 4), skipping the other bounding volumes.

4.4 SIGNED DISTANCE FUNCTION

A signed distance function (SDF) represents the shortest distance from a given point $\vec{x}$ to the boundary of a shape or region Ω , with a sign that indicates whether the point is inside or outside the region. If $\vec{x}$ is inside Ω , the function returns a positive value, decreasing to zero at the boundary. If $\vec{x}$ is outside Ω , the function returns a negative value. Some conventions reverse this, making the inside values negative and the outside values positive. This function is widely used in computer graphics, physics simulations and computational geometry to represent shapes, detect collisions and perform fluid and material simulations efficiently.

According to (Osher; Fedkiw, 2003), given a region Ω in a metric space and a point $\vec{x}$, while $\partial\Omega$ represents the boundary of Ω , the signed distance function $\phi(\vec{x})$ is defined as

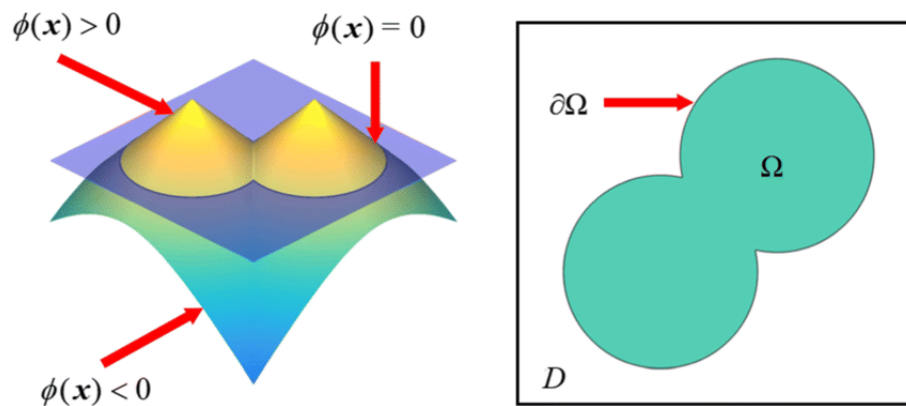
$$\phi(\vec{x}) = \begin{cases} d(\vec{x}), & \text{if } \vec{x} \in \Omega^+ \\ -d(\vec{x}), & \text{if } \vec{x} \in \Omega^- \end{cases}, \quad (2)$$

where:

$$d(\vec{x}) = \min(|\vec{x} - \vec{x}_l|) \text{ for all } \vec{x}_l \in \partial\Omega.$$

Figure 14 complements the mathematical definition of SDF demonstrating how the function's value and sign correspond to the spatial relationship between a point and the boundary of a region. Being the yellow/orange region the representation of the points that are inside the region Ω , the blue plane showing the points on the boundary and the green region representing the points outside of Ω .

Figure 14 – An example of a level set model with the signed distance function.



Source: (Song *et al.*, 2020)

4.5 PROFILING AND PERFORMANCE ANALYSIS

Profiling refers to the analysis of a software program's behavior while it is executing on a specific processor or platform (Tong; Khalid, 2007). Its primary goal is to identify computational bottlenecks, allowing developers to optimize the code and enhance overall performance. Various profiling tools are available, each employing different measurement techniques and gathering a wide range of performance data.

4.5.1 Overhead

A key factor in performance measurement is overhead, the additional execution time introduced by the profiling process compared to running without any measurements (Kaufmann, 2022). This overhead can slow down the system and affect measurement accuracy. When tracking the duration of tasks with different execution times, the impact of overhead varies. A long-running task experiences a minor relative increase in execution time, whereas a short task is affected more significantly. If a short task runs frequently, the accumulated overhead can distort the overall system behavior by inflating the time spent on that task. Completely eliminating overhead is impossible, as even the most precise measurements introduce slight delays. However, keeping overhead minimal is essential to reducing system perturbation and obtaining reliable performance data (Kaufmann, 2022).

4.5.2 Sampling

Sampling profilers operate by periodically probing a program's call stack at fixed intervals using operating system interrupts. Unlike tracing, which captures every event, sampling sacrifices accuracy to reduce overhead, allowing the program to run at near full speed (Kaufmann, 2022). The overhead of sampling depends on the sampling frequency and the time required to execute the measurement code. Since the measurement is performed asynchronously, sampling does not capture discrete events as tracing does but instead records states that persist for a certain duration.

To implement sampling, a mechanism is needed to trigger the measurement code at regular intervals, typically achieved using timer interrupts. However, because the program may be modifying its state while the measurement code reads it, synchronization is necessary to prevent data races. This is handled by briefly pausing execution, collecting the required data and then resuming normal operation. One common approach to minimize overhead is call stack sampling, which utilizes return addresses stored on the stack by the CPU when functions are invoked. By traversing these addresses, a snapshot of the program's execution state can be reconstructed without significantly impacting performance.

4.6 CONCURRENT AND PARALLEL COMPUTING

Concurrency occurs when two or more tasks start, run and complete within overlapping time periods, whereas parallelism refers to tasks that execute simultaneously on multiple CPUs or cores (Montez, 2024).

Process

A process is a program in execution, consisting of its code, execution state (program counter and registers) and memory sections (Silberschatz; Galvin; Gagne, 2012). These include the stack for temporary data, the data section for global variables, and the heap for dynamic memory allocation. Together, these elements define the process's execution context.

Threading

Threading is a technique in computer programming that allows a process to execute multiple tasks concurrently within the same memory space. A thread is the smallest unit of execution that an operating system can schedule independently (Silberschatz; Galvin; Gagne, 2012). Threads within the same process share resources such as memory, file handles and global variables, making communication between them efficient. However, because they share the same address space, improper synchronization can lead to issues such as race conditions and deadlocks.

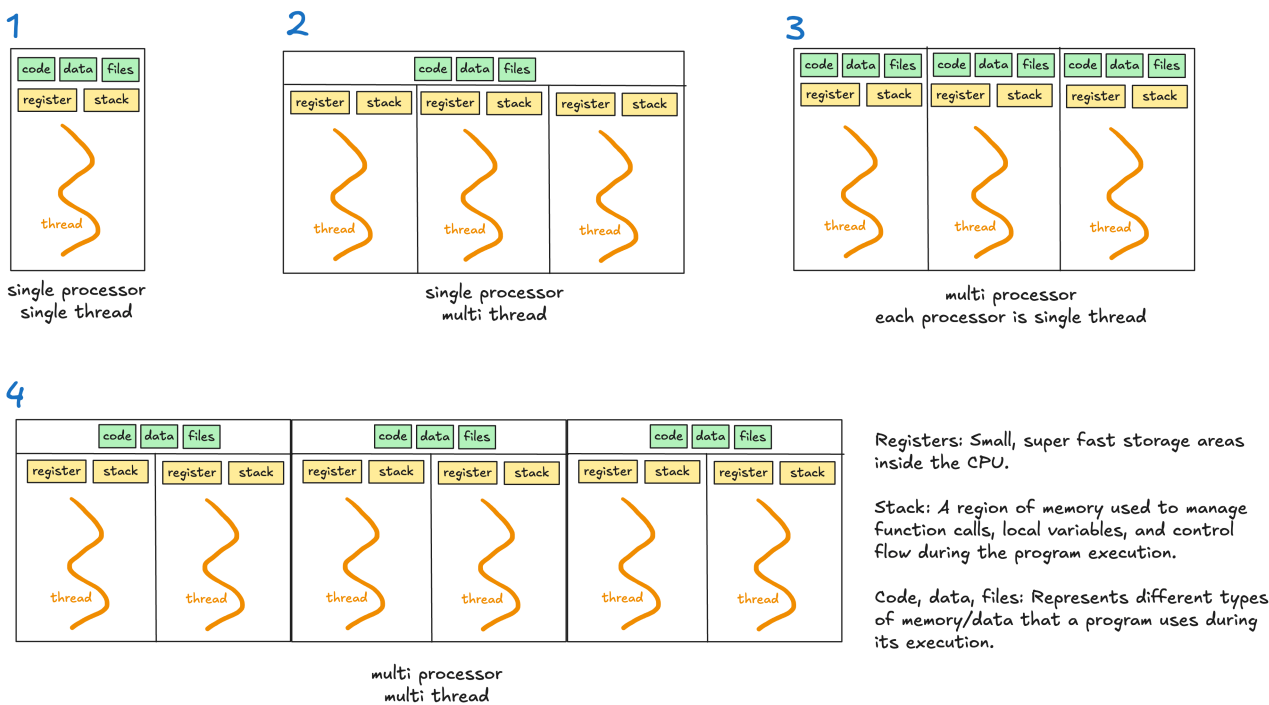
Multithreading

Multithreading refers to the ability of a program to execute multiple threads simultaneously, improving responsiveness and performance, especially on multi-core processors (Wong, 2025). By breaking tasks into smaller threads that run concurrently, multithreading can enhance efficiency in applications such as web servers, graphical user interfaces (GUIs) and real-time systems. However, managing multiple threads requires careful synchronization using mechanisms like mutexes, semaphores and condition variables to avoid conflicts when accessing shared resources.

Multiprocessing

While multithreading allows multiple threads to run within a single process, multiprocessing takes a different approach by running multiple processes in parallel, each with its own memory space (Andrews; Schneider, 1983). This method is particularly useful for computationally intensive tasks that benefit from full CPU utilization. Unlike threads, processes do not share memory by default, reducing the risk of synchronization issues but increasing communication overhead.

Figure 15 – Diagram illustrating different execution models: single processor single thread, single processor multi-thread, multi-processor single thread and multi-processor multi-thread, along with key memory components.



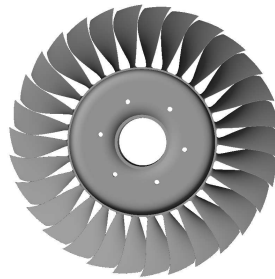
Source: (Chong, 2025)

5 TESTING AND BENCHMARKING

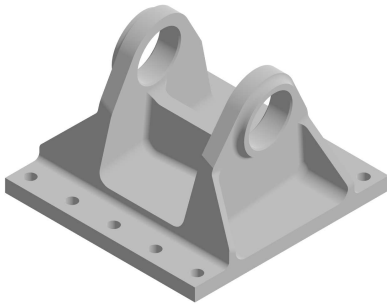
This chapter details the testing and benchmarking process used to evaluate various software solutions. It begins by introducing the standard test objects and a baseline analysis of the existing proprietary software is presented, establishing key performance metrics. Following this, the chapter describes the setup of the test environment and the investigation and selection of open-source libraries, setting the stage for a direct comparison of their capabilities and efficiency.

5.1 TEST OBJECTS

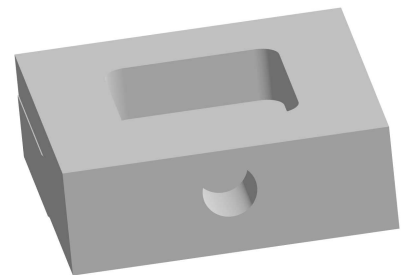
Figure 16 – Finish parts for the tested meshes.



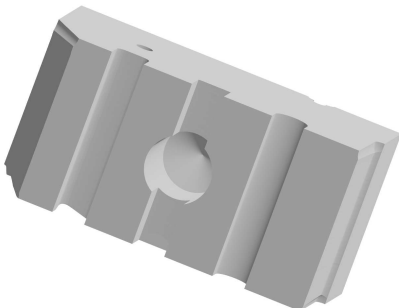
(a) DemoBlisk.



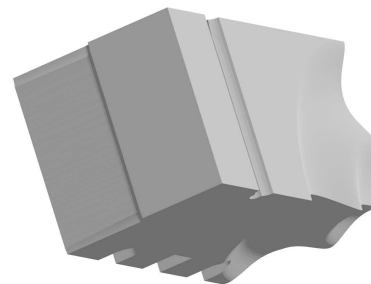
(b) Jet Bracket.



(c) PSM Demonstrator.



(d) Test Brick (Front view).



(e) Test Brick (Rear view).

Source: Author's archive.

The 3D models depicted in Figure 16 represent the 'finish parts' that serve as the reference geometries for the simulations and benchmark tests conducted throughout this project. These test objects vary in complexity and origin. The Alpha workpiece, for instance, is a complex industrial component provided by a client. To adhere to confidentiality requirements, its specific details and images are not included in this document. It serves as a representative test case for a real-world manufacturing part. The Demo Blisk is representative of critical components used in the aerospace industry, particularly in modern jet engines; it is valued for its high strength-to-weight ratio and complex geometry which enhances engine efficiency and performance. Similarly, the Jet Bracket is another component commonly found in aerospace applications, typically designed to support or connect various elements within an aircraft structure or engine assembly, often requiring high precision and specific material properties. Finally, both the Test Brick and the PSM Demonstrator are utilized primarily as internal demonstration and test geometries within the company, serving as standardized, familiar models for evaluating and validating various aspects of the software and algorithms under development.

Aligning with the standard data representation used across the company's simulation and analysis workflow, all test objects in this study are represented and processed exclusively as triangle meshes. This means that each model is composed of triangular faces and vertices, ensuring consistency across geometric processing and numerical methods. The polygon count for each object is as follows:

- **Alpha Workpiece:** 4.513.228 vertices, 2.051.262 faces
- **Demo Blisk:** 13.176.684 vertices, 4.449.802 faces
- **Jet Bracket:** 926.705 vertices, 559.662 faces
- **Test Brick:** 586.049 vertices, 200.618 faces
- **PSM Demonstrator:** 8532 vertices, 7388 faces

5.2 CURRENT CODE ANALYSIS

The existing proprietary software works as a multi-stage workflow for calculating deviation. The process begins by identifying and loading the necessary input files from a specified directory. The system performs initial checks to ensure all required files are present before proceeding.

The core of this software's methodology is its use of an internal precision simulation setting. Instead of directly comparing two triangle meshes, the system first converts the process mesh into a volumetric representation. This is done by creating a 2D grid and casting parallel rays through it. Each ray records the points where it enters and exits the 3D model. The collection of these depth recordings forms a volumetric grid that represents the shape of the process part.

The fineness of this grid and thus the precision of the simulation, is controlled by internal precision setting. A lower value for this parameter creates a finer grid, resulting in a more accurate and faithful volumetric representation of the part. This increased fidelity, however, comes at a significant computational cost, demanding more memory to store the larger grid and more processing time for the calculations. This presents a direct trade-off: while the system offers the flexibility to generate deviation maps at various quality levels by adjusting this internal parameter, the achievable precision is ultimately constrained by the available hardware resources.

After the calculation is complete, the system processes these results. It determines the maximum excess and minimum gouge values and, based on user-defined tolerances, assigns an overall quality status (e.g., Good, Medium, Bad) to the part.

Finally, the software generates the outputs. It applies a color map to the process mesh, where the color of each region corresponds to its calculated deviation value. This colored mesh is then exported as a new STL file for visualization.

To evaluate the performance impact of the internal precision setting, tests were run on several workpieces at different values. The results shown in Tables 1, 2, and 3 represent the mean and max values calculated from 10 executions for each test case. This 10-run method was used for all performance benchmarks in this study to ensure reliable and consistent metrics.

Table 1 – Tests with Internal Precision Setting of 0.7.

Test Object	Avg. CPU (%)	Avg. Max CPU (%)	Max CPU (%)	Avg. Memory (MB)	Avg. Max Memory (MB)	Max Memory (MB)	Duration (s)
Alpha work-piece	75.84	99.32	100	3081.42	3263.35	3282.59	478.69
Demo Blisk	79.13	99.45	100	1346.82	1477.62	1485.89	47.59
Jet Bracket	65.66	99.51	100	1059.96	1314.46	1316.62	9.30
Test Brick	53.72	92.53	98.86	195.37	291.98	293.47	0.66
PSM Demo.	53.74	79.51	82.28	63.25	71.79	75.30	0.20

Source: Author's archive.

Table 2 – Tests with Internal Precision Setting of 0.5.

Test Object	Avg. CPU (%)	Avg. Max CPU (%)	Max CPU (%)	Avg. Memory (MB)	Avg. Max Memory (MB)	Max Memory (MB)	Duration (s)
Alpha work-piece	79.82	99.67	100	3439.11	3682.41	3715.20	879.99
Demo Blisk	79.75	99.79	100	1468.39	1602.86	1611.75	94.76
Jet Bracket	73.92	99.58	100	1164.73	1342.31	1357.24	14.74
Test Brick	55.11	94.51	98.24	217.05	294.41	294.79	0.88
PSM Demo.	59.52	87.30	97.69	73.70	86.38	91.13	0.38

Source: Author's archive.

Table 3 – Tests with Internal Precision Setting of 0.3.

Test Object	Avg. CPU (%)	Avg. Max CPU (%)	Max CPU (%)	Avg. Memory (MB)	Avg. Max Memory (MB)	Max Memory (MB)	Duration (s)
Demo Blisk	79.70	99.63	100	1790.97	2040.27	2150.74	248.45
Jet Bracket	81.42	99.61	100	1312.92	1395.96	1415.77	35.54
Test Brick	72.31	99.48	100	238.11	291.73	293.07	1.33
PSM Demo.	65.84	97.99	99.20	106.18	138.49	140.43	0.90

Source: Author's archive.

The data presented in Tables 1, 2 and 3 quantitatively confirms the trade-off between precision and performance in the system. A clear trend emerges: as the internal precision resolution value decreases, the demand for computational resources rises.

This effect is most pronounced on the larger and more complex workpieces. For the Alpha workpiece model, decreasing the resolution from 0.7 to 0.5 nearly doubles the execution time from 478 seconds to 880 seconds, while also increasing the memory footprint. This demonstrates a non-linear scaling of resource consumption with increased precision. In contrast, for the smaller models like the Test Brick and PSM Demonstrator, the changes in performance metrics between resolutions are less dra-

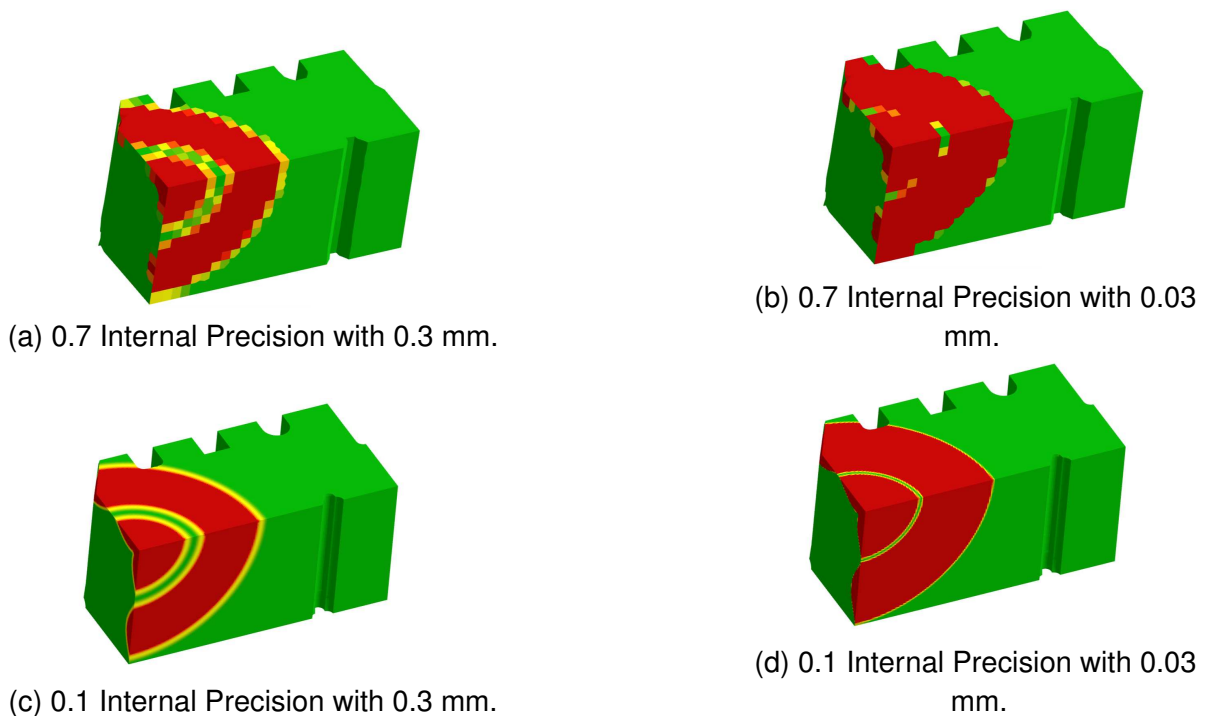
matic. This is because their simpler geometry does not generate a prohibitively large grid even at higher resolutions.

A critical limitation of this approach was observed during the 0.3 resolution test. The Alpha workpiece experiment is absent from Table 3 because the memory required to generate its high-resolution grid exceeded the 8 GB of RAM available on the test machine. This led to repeated application crashes, proving that the hardware specifications were insufficient for this level of precision on such a complex part.

It is also noteworthy that across all tests on larger models, the proprietary software consistently utilized the maximum available CPU power, with peak usage reaching 100%. This indicates that the proprietary software calculation is a highly intensive, parallelized process that will consume all available processing resources.

Figures 17, 18, 19 and 20 show the deviation maps generated by the proprietary software at different resolutions. These outputs serve as the visual benchmark for the open-source solutions and illustrate how the resolution setting impacts the final quality of the deviation map.

Figure 17 – Digital Twins generated by proprietary software with different Deviations for Test Brick.



Source: Author's archive.

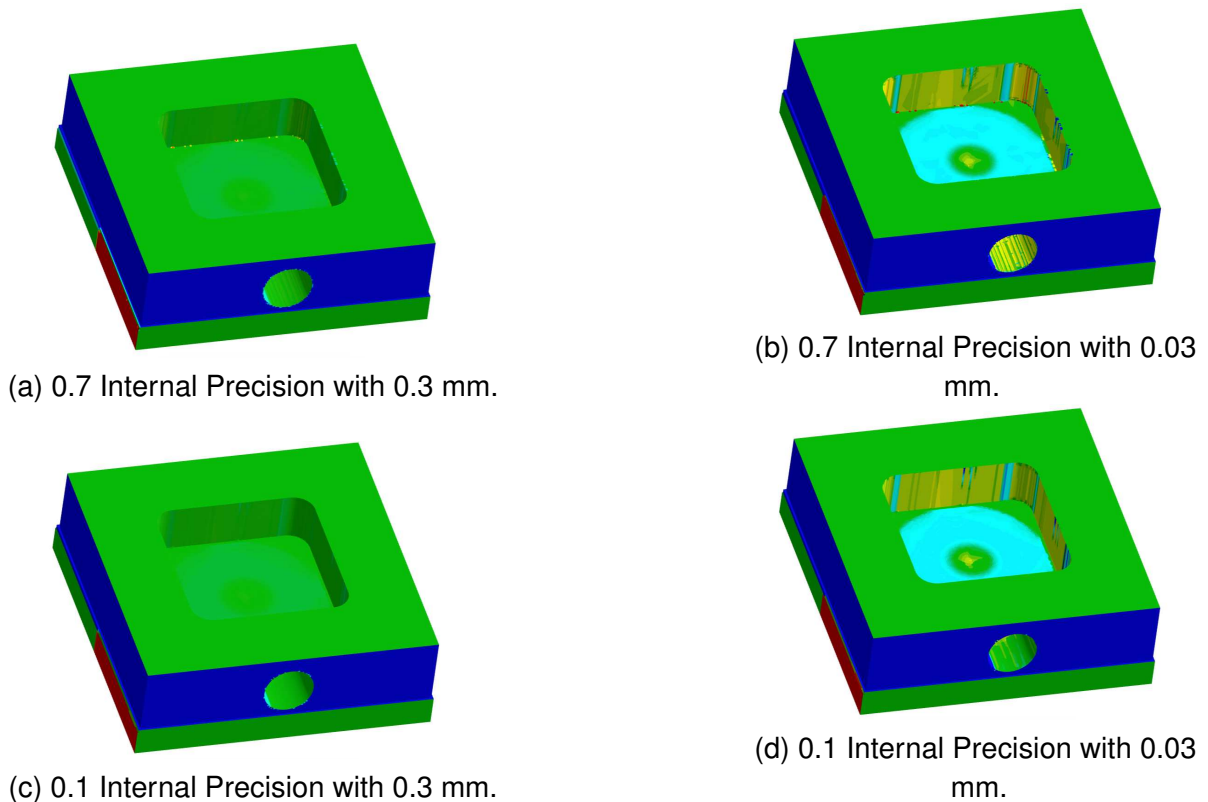
The deviation labels, such as 0.3 mm and 0.03 mm, refer to the symmetrical interval (e.g., $[-0.3, +0.3]$ mm) used for the color mapping of deviation values. A smaller interval results in a color map with higher sensitivity to smaller deviations.

Figure 17 demonstrates that for small workpieces, the method produces a pixelated or "blocky" deviation map. This is because the visualization directly reflects the

underlying grid, not a smooth per-vertex coloring. Consequently, achieving a visually useful result on small parts requires a very fine resolution to avoid excessive pixelation that could obscure the deviation details.

The PSM Demonstrator model illustrates the effect of the internal precision resolution on the final output. As shown in Figure 18, a coarse resolution of 0.7 at 0.03 mm only captures disconnected fragments of the central deviation. When a finer resolution of 0.1 is used, the same feature is correctly resolved as a complete, diamond-shaped area of darker blue. This highlights that a coarse grid can be insufficient to accurately represent the full geometry of a deviation.

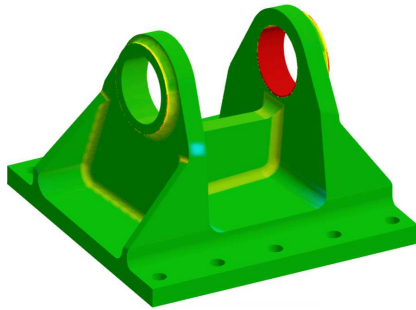
Figure 18 – Digital Twins generated by proprietary software with different Deviations for PSM Demo.



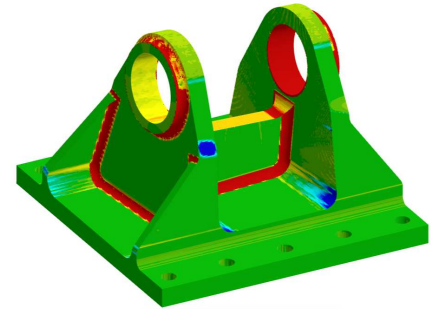
Source: Author's archive.

A similar analysis applies to the medium-sized Jet Bracket model. As shown in Figure 19, using a coarser resolution results in both incorrectly colored regions and a failure to accurately resolve the geometry of certain features when compared to the output from a finer resolution.

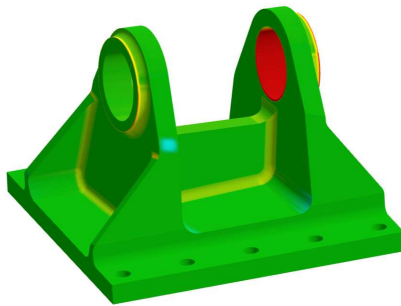
Figure 19 – Digital Twins generated by proprietary software with different Deviations for Jet Bracket.



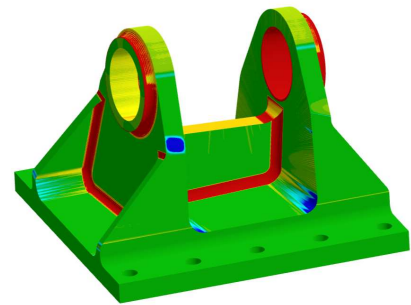
(a) 0.7 Internal Precision with 0.3 mm.



(b) 0.7 Internal Precision with 0.03 mm.



(c) 0.1 Internal Precision with 0.3 mm.

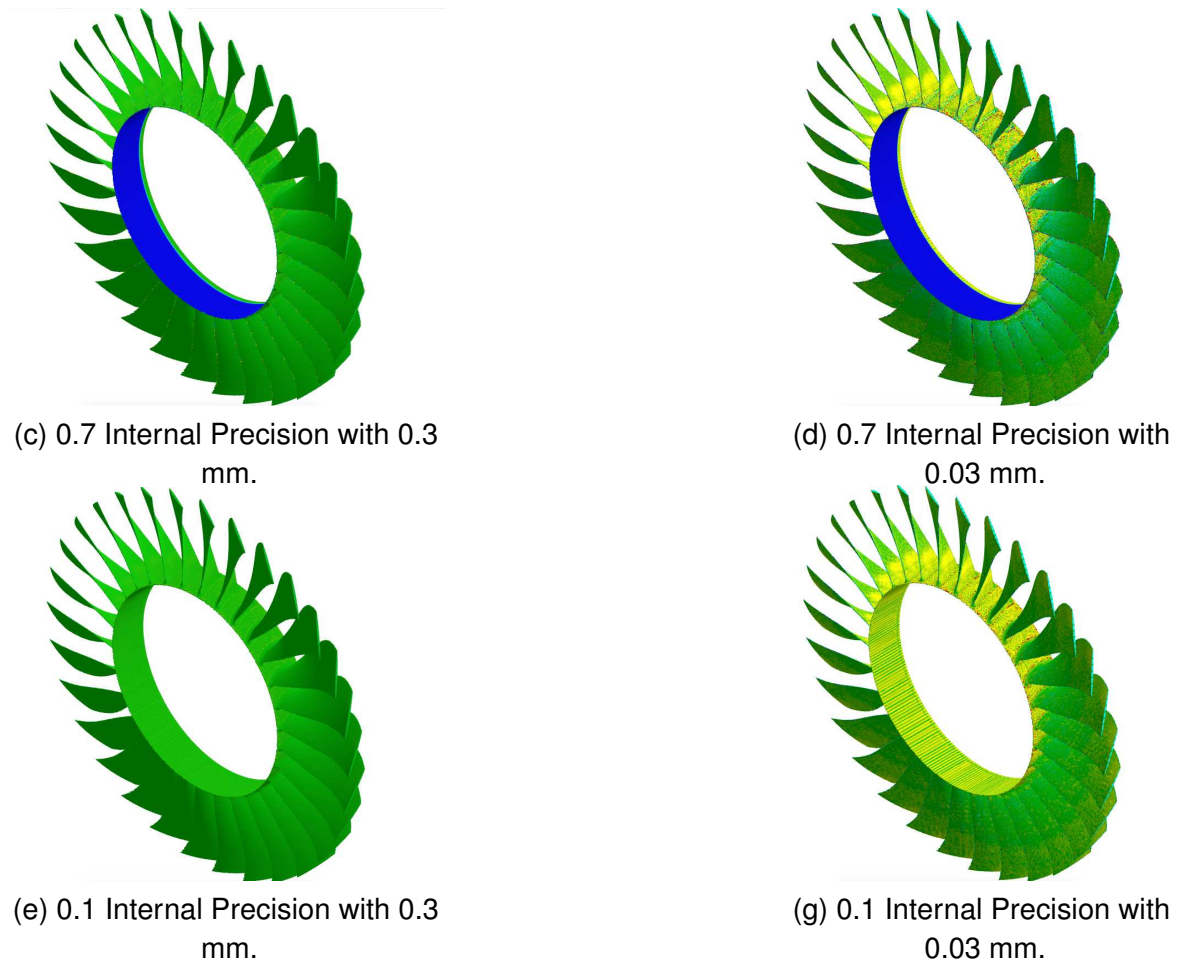


(d) 0.1 Internal Precision with 0.03 mm.

Source: Author's archive.

On the larger workpieces (Figure 20), the resolution-dependent differences are less pronounced but remain critical for manufacturing applications. The Demo Blisk test, however, reveals a major discrepancy: the central area is blue (negative deviation) at 0.7 resolution but green/yellow (zero/positive deviation) at 0.1 resolution. This is explained by a difference in the input data; the 0.1 resolution test was performed against a reference model that had a hole in its center.

Figure 20 – Digital Twins generated by proprietary software with different Deviations for Demo Blisk.



Source: Author's archive.

5.3 INVESTIGATION OF OPEN-SOURCE LIBRARIES

To achieve the objectives outlined in this project, an initial investigation into alternative open-source Python libraries was conducted. The primary goal of this investigation was to identify and benchmark potential candidates against the currently utilized library, aiming to determine which, if any, might offer superior performance, additional features, or better long-term maintainability for the project's specific needs. The selection process focused exclusively on Python libraries due to the language's accessibility, its ease of maintenance, rapid prototyping capabilities and the extensive ecosystem of scientific and data processing tools it offers.

A primary requirement for any considered library was functional parity with the core capabilities already employed. Candidates were evaluated based on their ability to fulfill several essential criteria:

- **Mesh Loading:** The capability to reliably load mesh data from *.stl* files.
- **Mesh Interaction:** The ability to access, iterate over and apply functions to the mesh data structures (vertices, faces, etc.).
- **Mesh Export:** Mandatory support for exporting processed meshes back into the *.stl* file format. Support for additional export formats was considered advantageous but not essential.
- **Mesh Processing:** Availability of fundamental mesh processing algorithms, particularly mesh decimation (simplification).
- **Distance Queries:** Functionality for performing various distance calculations, including point-to-point, point-to-surface and potentially point-to-mesh distances.

Beyond these core requirements, several desirable features were also considered during the evaluation, although their absence was not necessarily disqualifying:

- Support for importing or exporting CAD formats like STEP.
- Implementation of raycasting techniques for intersection analysis.
- Built-in visualization tools for rendering meshes and the results of calculations or transformations.

Furthermore, non-functional aspects played a crucial role in the selection process. Given the project's aim to rely on sustainable and freely usable tools, licensing was a primary concern. Only libraries distributed under permissive open-source licenses, specifically MIT or BSD licenses, were considered suitable to ensure compatibility with potential future uses and distributions of this project's work.

The overall health and viability of each library's open-source project were also assessed. This involved examining factors such as the level of recent development activity, the number and engagement of contributors, community feedback and user satisfaction (often indicated by metrics like repository stars) and the general credibility of the project. Particular attention was paid to the management of issues, analyzing the number of open issues relative to closed ones and the perceived responsiveness of the maintainers, as these factors indicate the library's stability and likelihood of future support. Libraries exhibiting signs of dormancy or having a large backlog of unaddressed critical issues were viewed with caution.

Frame 3 – Comparison of Open-Source Libraries.

Library	Load	Interact	Export	Process	Dist.	CAD	Raycast	Vis.	License	Contrib.	Stars	Issues
Open3D	+	+	+	+	+	-	+	+	MIT	240	12.200	1.200
Vedo	+	+	+	+	+	-	-	+	MIT	38	2.100	152
Trimesh	+	+	+	+	+	-	+	+	MIT	185	3.200	454
Pyvista	+	+	+	+	+	-	+	+	MIT	182	3.000	578
PyMesh	+	+	+	+	+	-	-	-	MIT	20	2.000	237
MeshLib	+	+	+	+	-	-	+	+	Custom	28	600	37
PyMeshLab	+	+	+	+	+	-	-	-	GPL	8	848	85

Source: Author's archive.

Based on the comparison presented in Frame 3, an initial filtering of the investigated libraries can be performed. Right away, MeshLib and PyMeshLab are discarded from further consideration due to licensing constraints. MeshLib uses a custom license and PyMeshLab uses the GPL license; neither meets the project's requirement for a permissive license (MIT or BSD) that allows for flexible integration and potential commercial application.

Another library, PyMesh, is also removed from the list of candidates despite fulfilling many of the functional requirements, including possessing the mandatory MIT license. The primary reason for its exclusion is its apparent lack of recent maintenance, with its last significant release dating back approximately five years. This inactivity raises concerns about its long-term viability, bug fixes and compatibility with newer versions of Python and its dependencies.

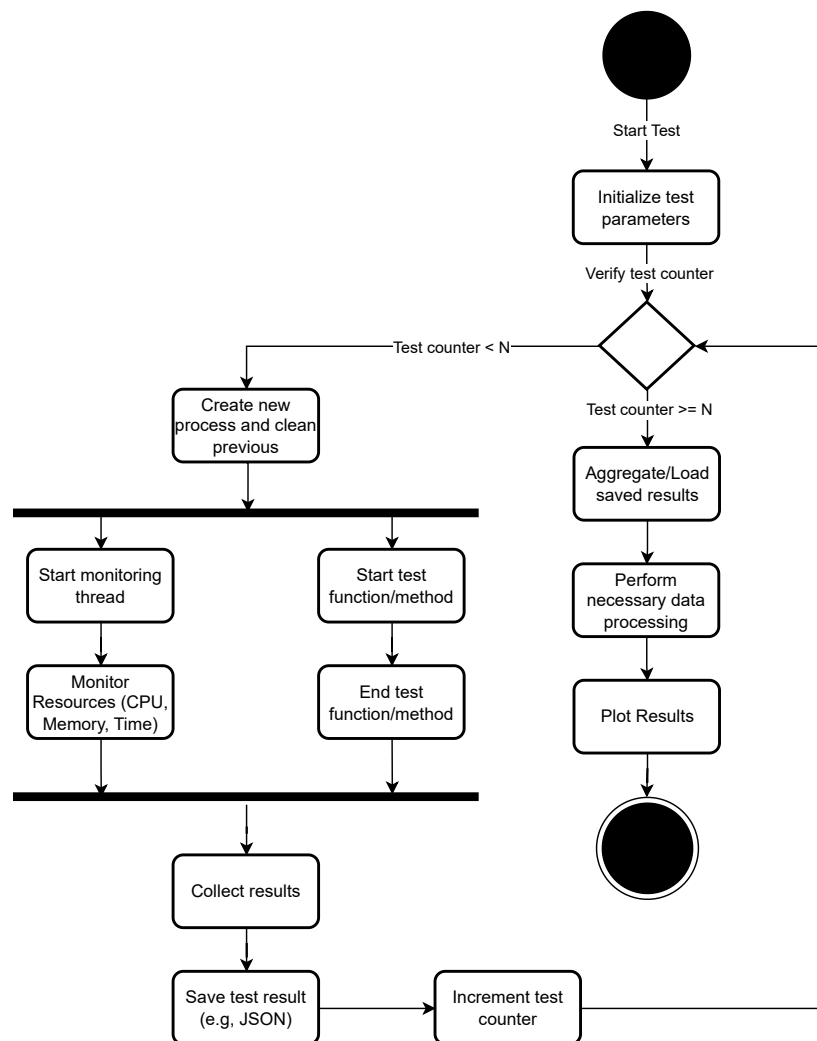
The remaining libraries (Open3D, Vedo, Trimesh and Pyvista) emerge as strong candidates for the subsequent benchmarking phase. As indicated in Frame 3, they all satisfy the mandatory requirements, such as *.stl* loading and exporting, mesh interaction and processing capabilities. Furthermore, they include several of the desirable optional features, like visualization tools and, in some cases, raycasting. Importantly, all four utilize the permissive MIT license. They also demonstrate positive indicators of project health, including substantial numbers of contributors and repository stars relative to their scope, active development (inferred from contribution frequency and recent releases, though not explicitly in the table) and a seemingly manageable ratio of open issues compared to community size.

Therefore, the libraries shortlisted for detailed performance benchmarking in this work are Open3D, Vedo, Pyvista and Trimesh. It should be noted, however, that this selection is based on the initial investigation; should significant inconsistencies, performance bottlenecks, or major integration challenges arise during the testing phase or the implementation, any of these candidates may still be discarded.

5.4 TEST ENVIRONMENT SETUP

This section describes the data acquisition process for testing library functions/methods, author-developed algorithms and custom code. Figure 21 presents an activity diagram illustrating the testing workflow implemented in this project.

Figure 21 – Proposed approach for evaluating the performance of test objects.



Source: Author's archive.

The testing procedure, as illustrated in Figure 21, commences with the initialization of necessary test parameters. It then enters a loop designed to run a specific number of tests, N. Before each test iteration, a verification step checks if the test

counter has reached N. If not ($\text{Test counter} < N$), the process involves cleaning up any resources from the previous test run and creating a new, isolated process for the current test.

The creation of separate processes for each test run leverages Python's *multiprocessing* library. This choice is significant, particularly in the context of Python's Global Interpreter Lock (GIL). The GIL is a mutex within the standard *CPython* interpreter that restricts execution, allowing only one thread to hold control of the Python interpreter and execute Python bytecode at any given moment. While this simplifies memory management and C library integration, it effectively prevents threads within the same process from achieving true parallelism on multi-core systems for CPU-bound tasks. By using *multiprocessing*, each test runs in its own distinct process with its own Python interpreter and memory space, thus bypassing the limitations imposed by the GIL within a single process and enabling true parallel execution if the underlying hardware supports it. For managing these processes efficiently, especially when running the same test function multiple times (potentially with different inputs), the *multiprocessing.Pool* class is utilized. A pool manages a predefined number of worker processes, distributing tasks among them and simplifying the collection of results, which is ideal for the repetitive nature of executing N tests. An example of a process creation using the *multiprocessing.Pool* class can be seen in Code Snippet 5.1:

Code Snippet 5.1 – Function to run the export performance test.

```
1  def run_trial_in_process(mesh_path, output_base, export_format):
2      """
3      Runs a single trial of the export performance test in an isolated
4      process.
5      """
6      with multiprocessing.get_context("spawn").Pool(1) as pool:
7          result = pool.apply(measure_open3d_export_performance, (mesh_path,
8                          output_base, export_format))
9      return result
```

The *run_trial_in_process* function demonstrates the instantiation and execution of a single test run. In this example, it utilizes a process pool to invoke a function that measures the export performance of a specific library, subsequently returning the measurement result.

Concurrently with starting the main test function within the newly created process, a dedicated monitoring thread is initiated. This thread is built using Python's standard *threading* library. Unlike the main test process, the monitoring task is typically I/O-bound (waiting for measurement intervals) rather than CPU-bound. Therefore, using a thread is appropriate here; it allows the monitoring to run concurrently without the overhead of a full process and its operation is not significantly hampered by the GIL because it spends much of its time waiting.

The monitoring thread employs the *psutil* library to gather resource utilization data. *psutil* (python system and process utilities) is a cross-platform library useful for retrieving information on running processes and system utilization, including CPU, memory, disk and network stats. Its key advantages include broad platform compatibility (Linux, Windows, macOS, etc.) and the ability to provide detailed information akin to standard command-line system tools. While potential minor drawbacks could include slight variations in metric definitions across platforms or the inherent overhead of any monitoring tool, *psutil* is widely chosen for its robustness and ease of use.

Code Snippet 5.2 – Function to monitor resources using psutil.

```
1  # Function to monitor CPU and memory usage continuously
2  def monitor_resources(resource_log, stop_event, interval=1):
3      process = psutil.Process()
4      while not stop_event.is_set():
5          cpu = process.cpu_percent()
6          memory = process.memory_info().rss / 1024**2 # Memory in MB
7          resource_log['cpu'].append(cpu)
8          resource_log['memory'].append(memory)
9          time.sleep(interval)
10
```

As shown in the Code Snippet 5.2, the monitoring thread instantiates a *psutil.Process* object to represent the current process. Within a loop controlled by a *stop_event*, it periodically calls *process.cpu_percent* to get the CPU usage percentage and *process.memory_info.rss* to retrieve the Resident Set Size (RSS) - the portion of memory occupied by a process that is held in RAM - which is then converted to megabytes. These CPU and memory readings are appended to a shared *resource_log* dictionary. The *time.sleep(interval)* call ensures measurements are taken at specified intervals; this periodic measurement constitutes a form of sampling profiling, as discussed in the theoretical background, capturing snapshots of resource usage over time.

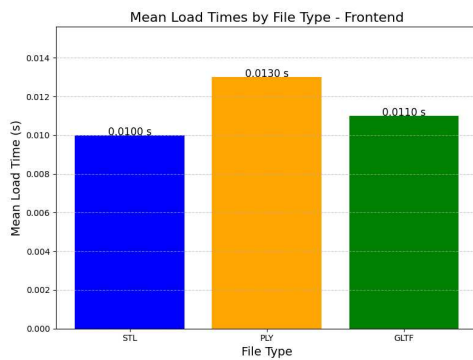
While the monitoring thread collects resource data, the main test function or method executes within its dedicated process. Once the test function completes, the monitoring thread is signaled to stop and the results from the test function itself are collected. These results, along with the resource usage data logged by the monitoring thread, are then saved, often in a structured format like JSON. The test counter is incremented and the loop continues until N tests have been completed.

After the loop finishes (Test counter $\geq$ N), the saved results from all test runs are aggregated or loaded. Finally, necessary data processing is performed on the aggregated results and plots are generated to visualize the findings.

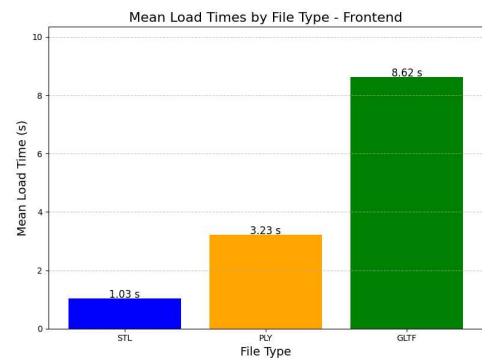
5.5 FRONTEND PARSING

In order to explore the feasibility of parsing different file formats in the frontend and potentially changing the current workflow, a simple test was conducted. This test compared the loading times of STL, PLY and GLTF formats using three different geometries. Each test ran 10 times to determine the mean loading time. The three geometries used were the PSM Demonstrator, Alpha workpiece and the Demo Blik:

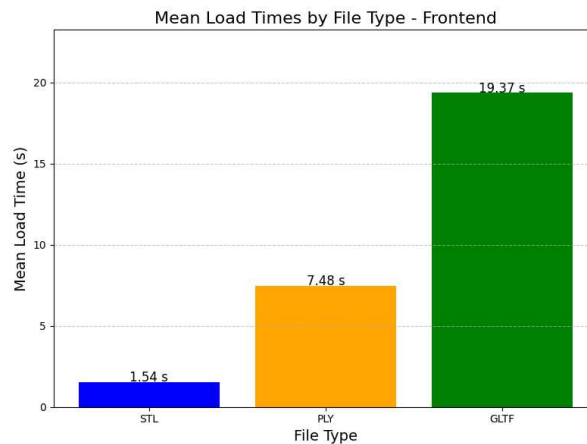
Figure 22 – Mean loading times of different file formats.



(a) PSM Demonstrator.



(b) Alpha workpiece.



(c) Demo Blik.

Source: Author's archive.

The significant variation in loading times between STL, PLY and GLTF files for the same geometric data arises from their fundamental differences and the complexity of their respective loaders. STL is the simplest, primarily handling raw mesh data. PLY adds some flexibility like color but remains mesh-centric. GLTF, in contrast, is a comprehensive scene description standard.

Consequently, Three.js loaders for these formats differ greatly. *STLLoader* and *PLYLoader* mainly parse geometry (vertices and faces) to build a *BufferGeometry*. While PLY might handle extra attributes or encodings, their task is relatively focused.

The *GLTFLoader* is far more complex, designed to parse scene structures (often

JSON), binary data buffers, detailed PBR materials, textures and reconstruct the entire scene hierarchy. Even for basic geometry, its architecture handles this full potential, increasing processing overhead compared to simpler loaders.

Performance results show a clear hierarchy: STL loads fastest, followed by PLY, with GLTF significantly slower, especially for complex models. Given the importance of frontend responsiveness, the loading time penalty of PLY and particularly GLTF is a major concern.

The findings strongly support using *STLLoader* for frontend geometry parsing to ensure the fastest loading times observed. Diversifying to include PLY or GLTF frontend parsing appears inadvisable due to their negative impact on performance and potential detriment to user interaction. Sticking with STL ensures optimal geometry loading speed in this context.

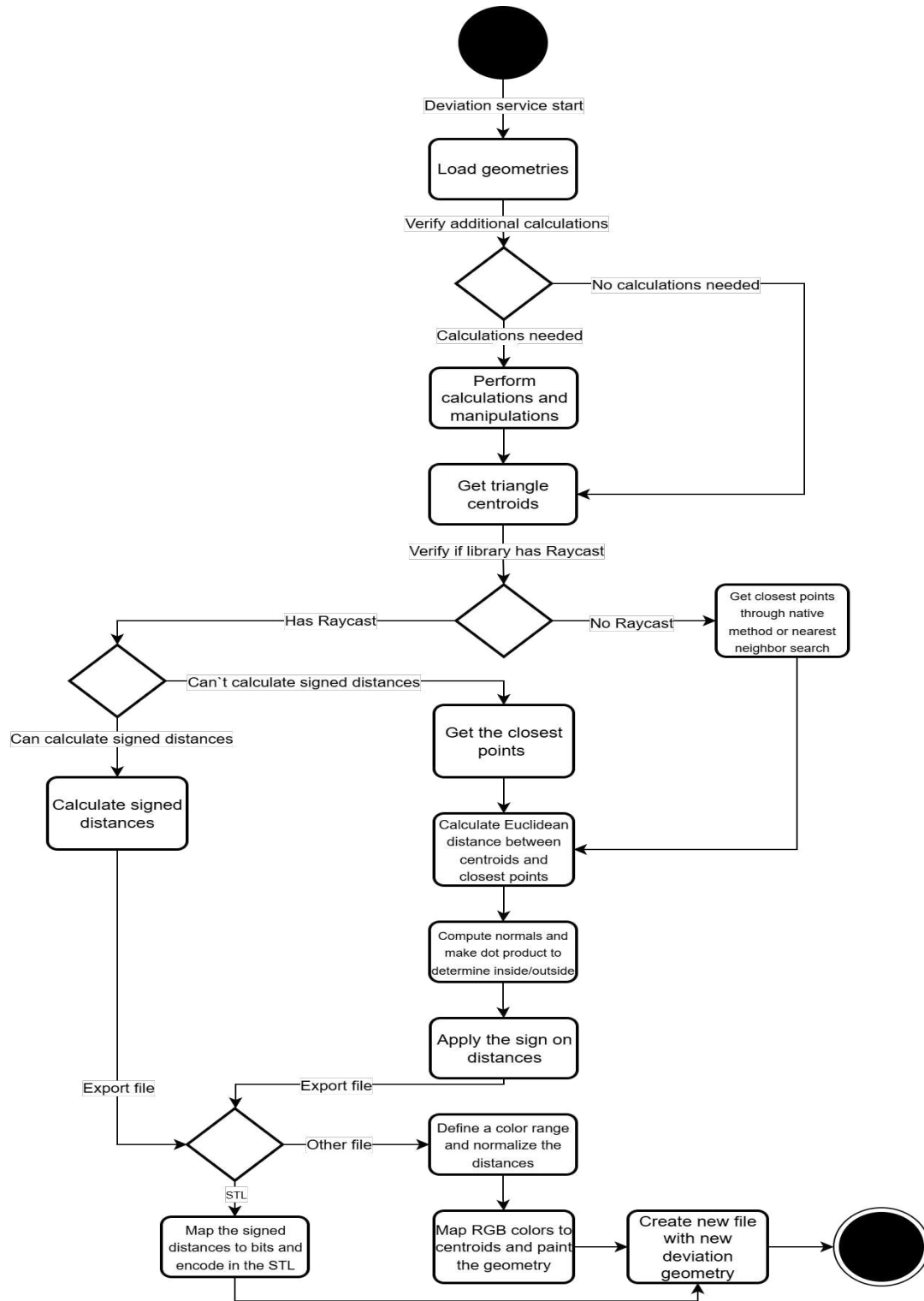
6 IMPLEMENTATION OF ALTERNATIVES

Since the libraries have been selected and the testing procedure established, the next step is to proceed with the actual implementation of the deviation calculation service. This stage focuses on integrating the chosen solutions into the system, ensuring efficient and accurate computation of deviation metrics.

6.1 ACTIVITY DIAGRAM

To structure both the deviation calculation process and the approach for using and comparing different libraries, an activity diagram has been developed, as shown in Figure 23. This diagram provides a visual representation of the computational workflow, guiding the implementation of various libraries.

Figure 23 – Proposed approach to evaluate the calculation of deviation for libraries.



Source: Author's archive.

The process commences from the initial node and progresses through distinct actions and decision points, outlining the major execution phases within the service. These phases ensure a systematic flow of data, starting from geometry loading and progressing through calculations, distance evaluation and final deviation visualization. Regardless of the specific library used, the fundamental structure remains consistent, with optimizations and modifications applied based on the capabilities of each method.

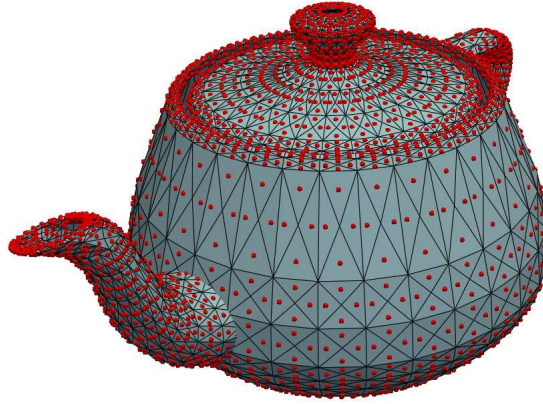
Each stage in the diagram represents a clearly defined step in the process, ensuring clarity and modularity. The blocks illustrate the key operations involved in deviation computation, while the arrows define the sequence in which these operations are performed.

The deviation calculation service begins with the loading of geometries, which is the first essential step in the workflow. At this stage, the system imports the necessary 3D models or datasets that will undergo deviation analysis. Specifically, two geometries need to be loaded: the model geometry, referred to as *finishPart*, which serves as the reference and the simulated part, referred to as *deviationMesh*, which is the geometry being analyzed for deviations. These geometries serve as the foundation for all subsequent calculations, ensuring that the required data is properly structured before proceeding further.

Once the geometries are loaded, the next step is to determine whether additional calculations or transformations are necessary. These operations are not dependent on the input data but rather on the requirements of the specific libraries being used. Some libraries require preprocessing steps before proceeding with the deviation calculations, such as modifying the structure of the loaded mesh to be compatible with their algorithms. Additionally, one of the alternative implementations, which combines two different libraries, involves converting a mesh format from one library into a format compatible with the other. If such adjustments are required, the system proceeds to the calculation and manipulation stage. If no such processing is needed, the workflow skips directly to getting the triangle centroids.

Following these preparatory steps, the algorithm extracts triangle centroids from the geometry. Using triangle centroids instead of vertices can simplify calculations by reducing the number of points to process, providing an average location for the triangle's deviation and providing a more general representation of a triangle's geometric center, especially when dealing with multiple triangles or complex shapes. Figure 24 illustrates an example of how the extracted triangle centroids are displayed in a mesh.

Figure 24 – Center of each triangle in a Teapot mesh.



Source: (Pyvista, 2025).

At this point, the system determines whether the chosen library supports raycasting, if it does support raycasting, the workflow proceeds with a direct distance calculation. However, if raycasting is unavailable, an alternative method is used, such as a nearest neighbor search or another native function to identify the closest points.

Once the closest points have been determined, the Euclidean distance between centroids and their respective closest points is computed. This step quantifies the deviation at each centroid location, providing a numerical representation of differences between the analyzed geometries. The Euclidean distance is computed as follows

$$d_i = \|C_i - P_i\| = \sqrt{(x_i - x'_i)^2 + (y_i - y'_i)^2 + (z_i - z'_i)^2}, \quad (3)$$

where $C_i = (x_i, y_i, z_i)$ represents the centroid of a given triangle and $P_i = (x'_i, y'_i, z'_i)$ is its closest point on the reference geometry.

After calculating raw distances, the system must determine whether the deviations are positive or negative in relation to a reference geometry. This is done by computing normals and performing a dot product operation, which helps classify points as either inside or outside the reference model. This classification is essential for correctly interpreting deviations and visualizing results meaningfully.

Given a triangular surface in the reference geometry, the normal vector n is usually obtained through

$$n = \frac{(v_2 - v_1) \times (v_3 - v_1)}{\| (v_2 - v_1) \times (v_3 - v_1) \|}, \quad (4)$$

with v_1 , v_2 and v_3 are the vertices of the triangle and $\times$ represent the cross product.

Now to get the direction of the vector it is necessary, for each centroid c_i , to subtract its closest point p_i :

$$v_i = c_i - p_i , \quad (5)$$

with this new vector, applying a dot product with the normal vector n_i at the closest point determines the sign of the deviation

$$s_i = v_i \cdot n_i , \quad (6)$$

with $s_i \geq 0$ representing when the centroid is "outside" or along the positive normal direction of the closest point on the reference surface, while $s < 0$ represents when the centroid is "inside" the reference geometry. So it is possible to define a SDF function d_i^{signed} , like seen in Equation 2, as:

$$d_i^{\text{signed}} = \begin{cases} d_i, & \text{if } s_i \geq 0 \\ -d_i, & \text{if otherwise} \end{cases} \quad (7)$$

With these signed distances computed, the sign is applied to the deviation values, ensuring that positive and negative deviations are correctly assigned. This step prepares the data for visualization and further analysis.

For reasons of confidentiality, the specific implementation details of the color mapping based on normalized deviation values and the direct encoding of signed distances into the STL file format have been omitted from this work.

6.2 OPEN 3D

The Open3D implementation utilizes the *RaycastingScene* class, this class applies raycasting to enable distance queries. Such queries treat the reference triangle mesh as an implicit signed distance function. This capability allows the system not only to query the shortest distance from a point to the surface of the mesh but also to determine if that point (such as a triangle centroid from the *deviationMesh*) lies inside or outside the reference mesh, thereby establishing the signed deviation.

Code for the Open3D implementation, along with the other libraries discussed in this chapter, has been omitted from this document for confidentiality reasons. However, the core methodology and algorithmic structure of each implementation will still be described.

The core logic is encapsulated in the *compare_and_color_mesh_open3d* function. This function begins by loading the *finish_part* and *deviation_mesh* using Open3D's tensor-based I/O. The *deviation_mesh* is then converted to its legacy format to facilitate the computation of triangle normals and the extraction of vertex and triangle data. From this data, the centroids of each triangle in the *deviation_mesh* are calculated.

These centroids are subsequently used as query points to determine their signed distance to the *finish_part*. This is achieved by calling the helper function *mesh_to_cloud_signed_distances*. Inside this helper, a *RaycastingScene* is initialized and the *finish_part* mesh is added to it. The *scene.compute_signed_distance* method is then invoked with the centroids. This method effectively treats the *finish_part* mesh as a closed surface and determines the sign of the distance by counting ray intersections from the query points (centroids), with negative distances indicating points inside the surface.

Table 4 summarizes the results obtained from executing this implementation. The tests were conducted following the methodology detailed in the activity diagram (Figure 21).

Table 4 – Open3D metrics.

Test Object	Avg. CPU (%)	Avg. Max CPU (%)	Max CPU (%)	Avg. Memory (MB)	Avg. Max Memory (MB)	Max Memory (MB)	Duration (s)
Alpha work-piece	8.58	81.33	91.01	2338.57	7371.74	7375.51	36.65
Demo Blisk	8.98	78.88	90.78	3219.84	8823.78	9121.34	29.96
Jet Bracket	9.75	77.77	83.62	1029.75	3310.22	3320.81	14.38
Test Brick	15.47	81.23	85.77	271.46	537.52	549.94	2.78
PSM Demo.	32.25	72.38	83.11	150.59	157.66	165.97	0.28

Source: Author's archive.

With the quantitative performance metrics established, the analysis now shifts to a qualitative assessment of the implementation's output. This involves a visual inspection of the generated deviation maps to verify the accuracy of the color mapping across a range of values. Furthermore, this section will discuss any observed artifacts or potential sources of error in the final rendered geometry.

Figure 25 – Digital Twins generated by Open3D with different Deviations for Test Brick workpiece.



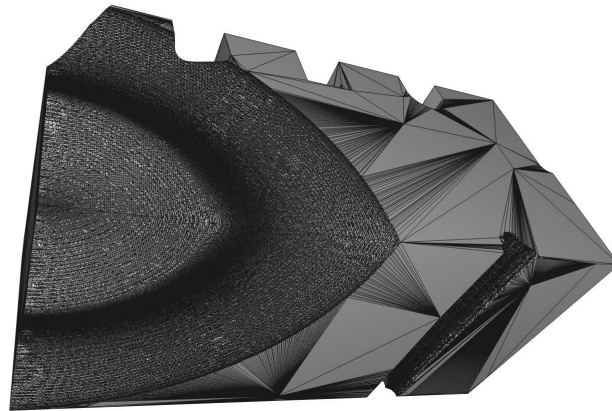
Source: Author's archive.

A direct comparison between the Open3D result (Figure 25) and the output from the proprietary software (Figure 17) reveals that at a coarse resolution of 0.3 mm, the deviation maps are visually consistent and correct. However, when the analysis is performed at a finer resolution of 0.03 mm, visual artifacts become apparent in the Open3D result. These artifacts manifest as erroneous lines and colored points that do not correspond to actual geometric deviations.

Further investigation indicates that this issue is not a flaw in the deviation calculation algorithm itself, but rather stems from pre-existing defects in the input STL file. The mesh contains topological errors, such as non-manifold geometry, where edges are improperly connected, for instance by being shared by more than two triangles instead of the standard two. A specific example of this is the presence of T-junctions, where a vertex from one triangle rests on the edge of an adjacent one without being properly joined.

At a fine resolution, the distance calculation algorithm can misinterpret these poorly connected regions, leading to the observed incorrect coloring. A wireframe visualization of the mesh, as seen in Figure 26, confirms this diagnosis, revealing the irregular and poorly structured triangulation in the exact locations where the artifacts appear. This highlights the critical importance of using high-quality, manifold meshes as input for accurate deviation analysis.

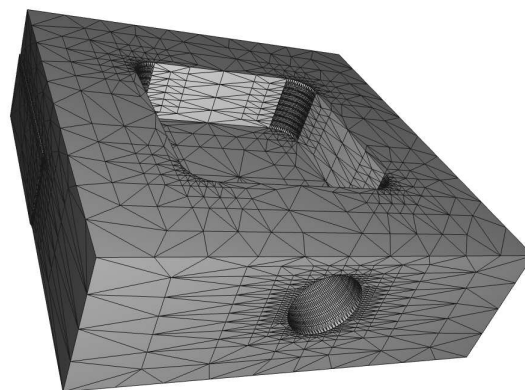
Figure 26 – Wireframe model of the Test Brick Finish Part.



Source: Author's archive.

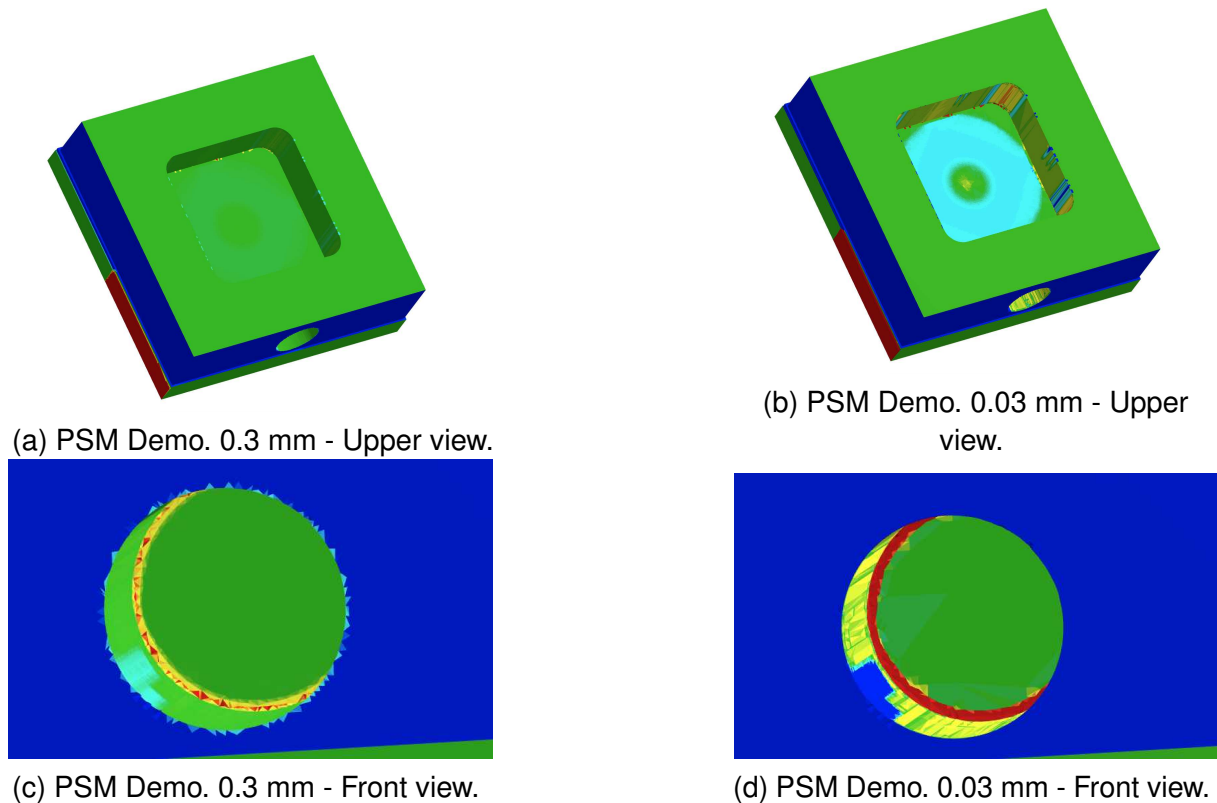
To corroborate the diagnosis, the results from the PSM Demonstrator model are now examined. Unlike the previous model, the PSM Demonstrator has a more well-structured, manifold mesh, as shown in Figure 27, than Test Brick.

Figure 27 – Wireframe model of the PSM Demo. Finish Part.



Source: Author's archive.

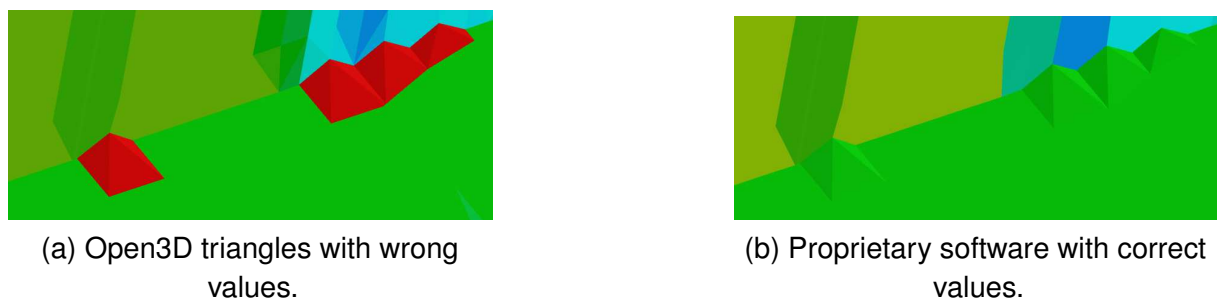
Figure 28 – Digital Twins generated by Open3D with different Deviations for PSM Demo. workpiece.



Source: Author's archive.

On initial inspection, the Digital Twin generated by the Open3D implementation appears identical to the output from the proprietary software. However, a more detailed analysis reveals localized inaccuracies. In certain regions of the mesh, the color mapping is incorrect, showing values that deviate significantly from the expected results established by the proprietary software benchmark. Figure 29 highlights these differences for a direct comparison.

Figure 29 – Direct Comparison between triangles and their deviation values.



Source: Author's archive.

Discussing with colleagues, including the developer of the original service, the root cause of these inaccuracies was identified. The issue does not lie in the new

deviation algorithm, but rather in the data generation process from the upstream service that provides the input STL file.

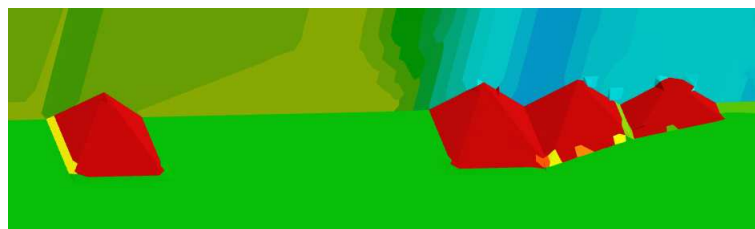
This upstream service performs its simulations using a proprietary data format, not STL files directly. The erroneous triangles are artifacts introduced during the conversion from this internal format to the STL file that is passed to the Deviation Service. The legacy proprietary system was designed to interpret and effectively ignore these specific artifacts during its analysis, a capability the new open-source implementation does not inherently possess.

To address this, two potential solutions were considered. The first involves modifying the upstream service to output a list of the artifact-producing triangles along with the STL file. The second involves creating a new interface, such as an API call, for the Deviation Service to query this information. However, both approaches would introduce significant latency and complexity into the data pipeline.

After consultation with the project supervisor and team, it was determined that implementing either solution would require substantial modifications to another service. Therefore, this task was deemed outside the scope of the current project due to its complexity and time requirements, as it would involve altering systems beyond the author's direct control and familiarity with its internal operations.

To support this explanation, the proprietary software was re-run with an internal precision resolution of 0.15. As seen in Figure 30, the output now more closely matches the Open3D result, though it remains lower in quality. This discrepancy is likely because the initial benchmark was generated at an even coarser resolution, causing its internal artifact handling to produce a visually different output.

Figure 30 – Proprietary software test with 0.15 internal precision resolution.



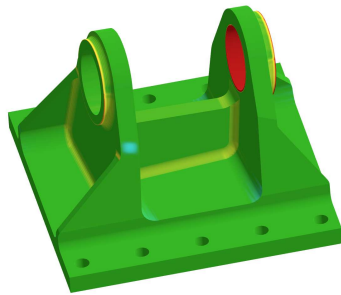
Source: Author's archive.

It is also noteworthy that these visual artifacts are more pronounced on smaller workpieces. This trend is evident when examining the results for the other simulated models in Figure 31.

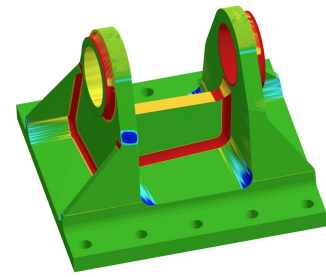
This phenomenon can be attributed to the issue of relative scale. On smaller geometries, a topological error of a fixed absolute size, such as a small gap or an improperly connected edge, represents a much larger fraction of a feature's total di-

mension. Consequently, it is more likely to be visually prominent and to be flagged by the high-resolution deviation analysis. Conversely, on larger workpieces with broader surfaces, the same absolute error may be negligible in comparison to the overall size and could fall below the threshold of visual or analytical significance, making it appear as if the issue does not occur.

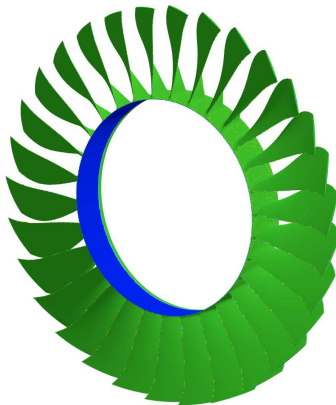
Figure 31 – Digital Twins generated by Open3D with different Deviations for the rest of the workpieces.



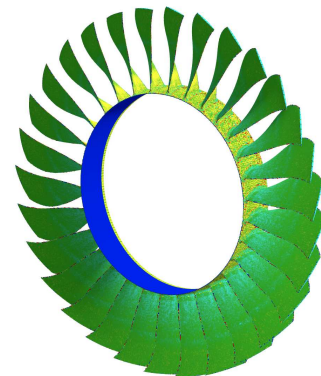
(a) Jet Bracket 0.3 mm.



(b) Jet Bracket 0.03 mm.



(a) Demo Blisk 0.3 mm.



(b) Demo Blisk 0.03 mm.

Source: Author's archive.

A comparison of the Digital Twins for the remaining workpieces (Figure 31) with those generated by the proprietary software reveals a high degree of correlation. This observation serves to corroborate two earlier conclusions: first, the phenomenon of relative scale, as these larger models do not exhibit the same artifacts; and second, the critical importance of a well-structured input mesh, as these models produce accurate results when their topology is sound.

Collectively, these findings demonstrate that the raycasting technique within Open3D is a robust and effective replacement for the current proprietary method. This conclusion is supported not only by its superior performance metrics but also by its ability to produce results of comparable, if not higher, fidelity, especially when considering that the proprietary system may operate at lower resolutions to mask mesh artifacts.

6.3 PYVISTA

The PyVista documentation provides an example of a ray tracing implementation for calculating surface-to-surface distances, which was initially considered for this project. The code for this method is included for reference in Annex A.1. The fundamental limitation of this documented method is its iterative nature: it requires looping through each query point individually to perform the ray intersection test and subsequent calculations. This lack of vectorization makes the process inherently unscalable for large point sets.

To mitigate this performance bottleneck, a parallel processing strategy was tested. Initial results were promising on small geometries like the PSM Demonstrator, which yielded performance metrics comparable to the Open3D implementation. However, the approach failed to scale effectively. When applied to even a moderately sized model like the Test Brick, the execution time became unacceptably long, comparable to the time taken by the entire proprietary software workflow on a much larger component. Given these critical performance limitations, this native ray tracing approach was deemed unsuitable for the project's requirements and was discarded from further analysis.

As an alternative approach to ray tracing, a specific filter was adopted that leverages PyVista's direct access to its backend. Visualization Toolkit (VTK) is a powerful, open-source C++ library for 3D computer graphics modeling, image processing, volume rendering, scientific visualization and 2D plotting. PyVista serves as a high-level Python wrapper for VTK, simplifying access to its performant algorithms.

The main logic for this implementation begins by loading the *finish_part* and *deviation_mesh* using PyVista's *read* function. The centroids of the *deviation_mesh* are then computed using the *cell_centers* method.

The next step is to find the closest point on the surface of the *finish_part* for each centroid. This is accomplished with the *find_closest_cell* method, which leverages VTK's *vtkCellLocator* in the backend. This locator builds a spatial data structure known as an octree, which recursively subdivides the 3D space occupied by the mesh into eight smaller cubic regions. This hierarchical structure allows the search algorithm to quickly discard large parts of the mesh that are far from a query point, significantly accelerating the search. This ensures an efficient search for the true closest point on the mesh surface, which may lie on a face or edge rather than just being the nearest vertex.

Once the closest points are identified, the raw Euclidean distance between each centroid and its corresponding closest point is calculated. To determine the sign of this distance, the surface normals of the *finish_part* are computed. The direction vector from each closest point to its centroid is then used in a dot product with the surface normal of the corresponding cell. This operation is performed efficiently for all points

at once using NumPy's einsum function. The notation 'ij,ij->i' specifies a vectorized dot product; for each point, it multiplies the corresponding direction and normal vectors element-wise and sums the result, producing a single array of scalar values.

The performance metrics for this implementation are detailed in Table 5:

Table 5 – Pyvista metrics.

Test Object	Avg. CPU (%)	Avg. Max CPU (%)	Max CPU (%)	Avg. Memory (MB)	Avg. Max Memory (MB)	Max Memory (MB)	Duration (s)
Alpha work-piece	8.20	11.06	11.20	1030.48	1582.84	1583.51	52.38
Demo Blisk	8.16	10.48	11.20	911.87	1179.41	1181.11	35.71
Jet Bracket	8.17	10.39	11.20	517.63	763.46	764.25	23.30
Test Brick	8.13	10.07	11.20	271.46	312.19	321.59	9.81
PSM Demo.	5.81	8.74	9.70	187.45	190.96	191.79	0.23

Source: Author's archive.

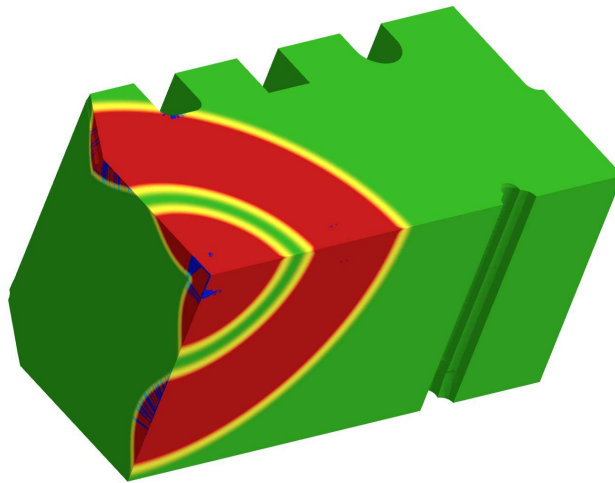
An examination of the Digital Twin generated by the PyVista implementation for the Test Brick workpiece (Figure 32) reveals visual artifacts. Specifically, clusters of blue triangles appear near sharp edges, indicating that a positive deviation is being incorrectly calculated as a large negative deviation.

The source of this discrepancy lies in the method used to determine the sign of the distance. The PyVista implementation first uses *find_closest_cell* to identify the nearest point on the *finish_part* surface and the specific cell that contains it. Subsequently, it uses the normal vector of that single, closest cell to perform the dot product calculation for the sign.

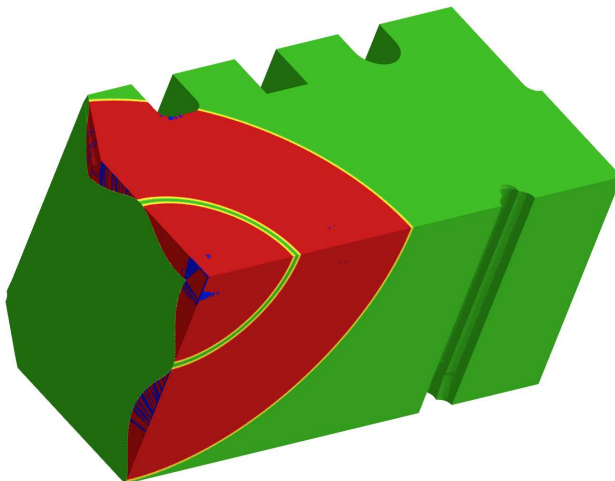
This approach can be unreliable in areas of high curvature, near sharp features, or at non-manifold edges. In such complex regions, the normal of a single cell may not accurately represent the true orientation of the overall surface at that location. An incorrectly oriented normal vector leads to an erroneous sign from the dot product calculation, causing the observed artifacts.

This issue is likely a combination of this specific method's limitations and the known topological flaws of the Test Brick mesh, as discussed in the Open3D analysis. It suggests that this PyVista implementation is sensitive to input data quality and may perform more reliably on larger, well-structured meshes.

Figure 32 – Digital Twins generated by Pyvista with different Deviations for Test Brick workpiece.



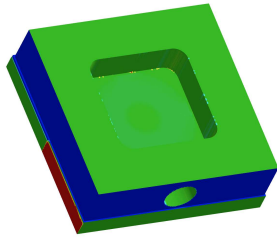
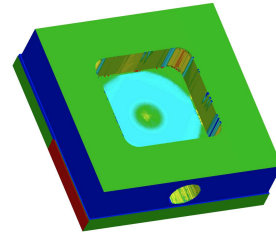
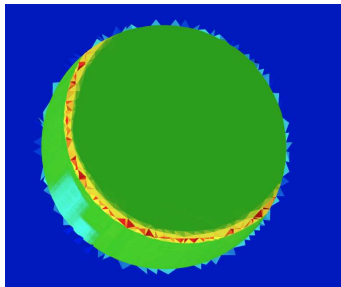
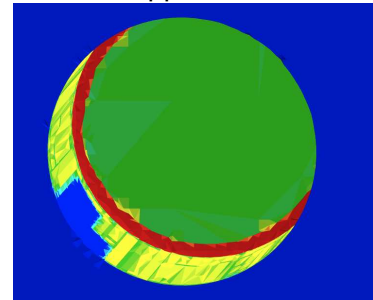
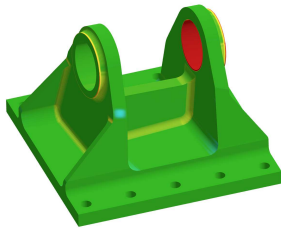
(a) Test Brick 0.3 mm.



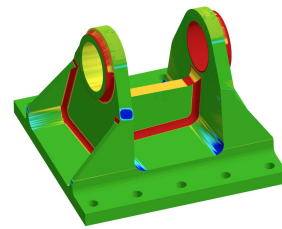
(b) Test Brick 0.03 mm.

Source: Author's archive.

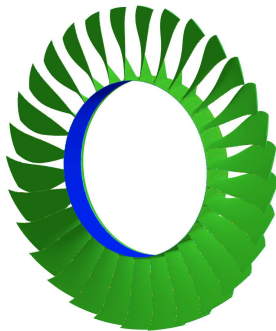
Figure 33 – Digital Twins generated by Pyvista with different Deviations.

(a) PSM Demo. 0.3 mm -
Upper view.(b) PSM Demo. 0.03 mm -
Upper view.(c) PSM Demo. 0.3 mm -
Front view.(d) PSM Demo. 0.03 mm -
Upper view.

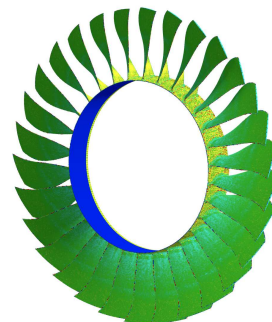
(e) Jet Bracket 0.3 mm.



(f) Jet Bracket 0.03 mm.



(g) Demo Blisk 0.3 mm.



(h) Demo Blisk 0.03 mm.

Source: Author's archive.

For the larger, well-defined workpieces, the PyVista implementation performs as expected. The results, presented in Figure 33, are accurate and align closely with both the Open3D and proprietary software outputs. This confirms that the method is reliable for high-quality meshes.

6.4 VEDO

The analysis now turns to the implementation using Vedo, a Python module designed for the scientific analysis and visualization of 3D objects and point clouds. Similar to PyVista, Vedo is built upon VTK, serving as a high-level interface to its powerful C++ algorithms.

Following the established activity diagram 23 the initial steps of the code remain the same by loading the meshes and then performing the right computations to it.

The core of the calculation is performed by the *distance_to* method, which internally calls VTK's *vtkDistancePolyDataFilter*. To find the shortest distances, this filter utilizes the same *vtkCellLocator* employed by the PyVista implementation. Furthermore, the process for determining the sign of the distance is conceptually the same as the manual method described previously, involving a dot product between direction vectors and surface normals. However, with Vedo, this entire operation is encapsulated and handled internally by the VTK filter when the *signed=True* parameter is set.

The performance metrics for this implementation are presented in Table 6.

Table 6 – Vedo metrics.

Test Object	Avg. CPU (%)	Avg. Max CPU (%)	Max CPU (%)	Avg. Memory (MB)	Avg. Max Memory (MB)	Max Memory (MB)	Duration (s)
Alpha work-piece	7.94	65.27	83.62	888.16	1317.43	1400.51	103.68
Demo Blisk	8.21	62.99	72.87	1099.53	1381.03	1404.98	60.21
Jet Bracket	8.10	56.43	69.84	441.56	684.06	724.11	30.36
Test Brick	8.32	65.06	80.49	248.17	289.17	295.10	23.78
PSM Demo.	10.59	62.69	82.29	137.77	190.01	194.19	3.24

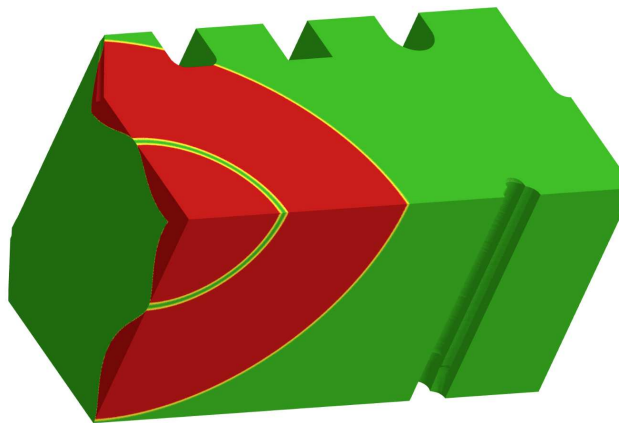
Source: Author's archive.

The visual output of the Vedo implementation is now examined, beginning with the Test Brick model shown in Figure 34.

Figure 34 – Digital Twins generated by Vedo with different Deviations for Test Brick workpiece.



(a) Test Brick 0.3 mm.



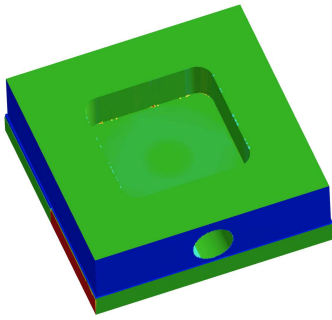
(b) Test Brick 0.03 mm.

Source: Author's archive.

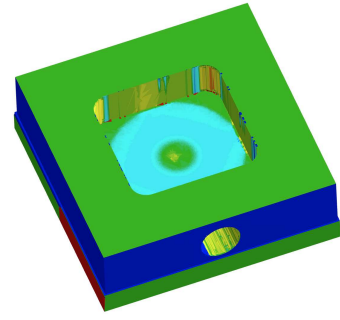
For the Test Brick workpiece, the Vedo implementation produces a clean result, free from the artifacts seen in both the PyVista and Open3D outputs. Although Vedo and PyVista use the same underlying distance calculation method, Vedo's internal process for signing the distances proves more robust for this flawed, small-scale mesh. Furthermore, its ability to correctly handle the same topologically flawed data that caused errors in the Open3D implementation demonstrates that Vedo's internal methods are sufficient to manage imperfect mesh data, at least for smaller geometries and achieve the quality required by the service.

The results for the remaining workpieces are presented in Figure 35.

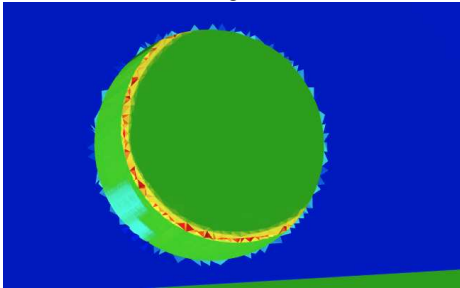
Figure 35 – Digital Twins generated by Vedo with different Deviations.



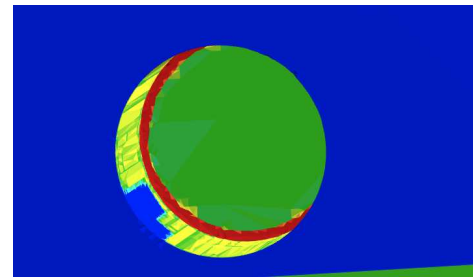
(a) PSM Demo. 0.3 mm - Upper view.



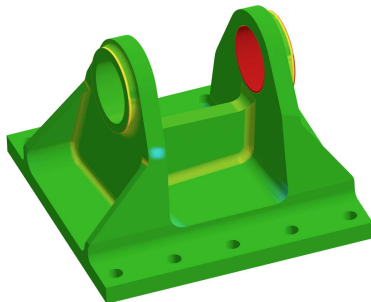
(b) PSM Demo. 0.03 mm - Upper view.



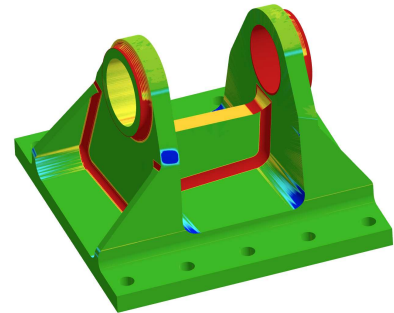
(c) PSM Demo. 0.3 mm - Front view.



(d) PSM Demo. 0.03 mm - Front view.



(a) Jet Bracket 0.3 mm.

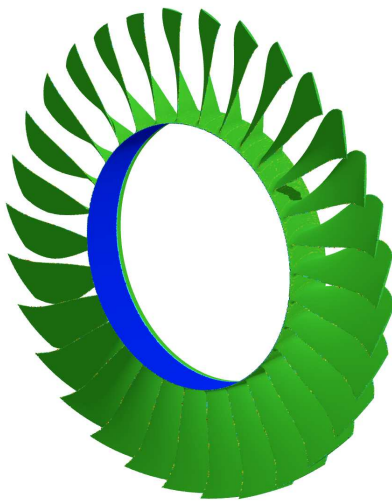


(b) Jet Bracket 0.03 mm.

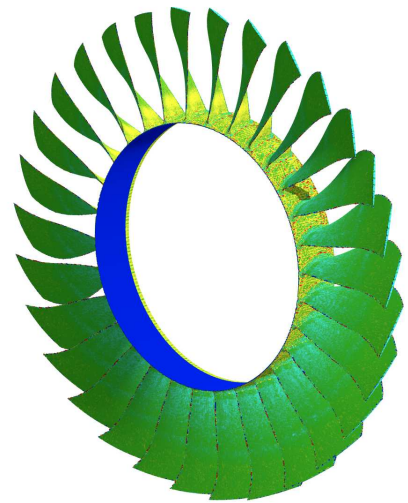
Source: Author's archive.

The results for the PSM Demonstrator, Jet Bracket and Alpha workpiece are identical between the Vedo and PyVista implementations, which validates their shared VTK-based distance query method. For these well-defined meshes, the output also aligns closely with the proprietary software, confirming the accuracy of the open-source approach on high-quality data.

Figure 36 – Digital Twins generated by Vedo with different Deviations for Demo Blisk workpiece.



(a) Demo Blisk 0.3 mm.

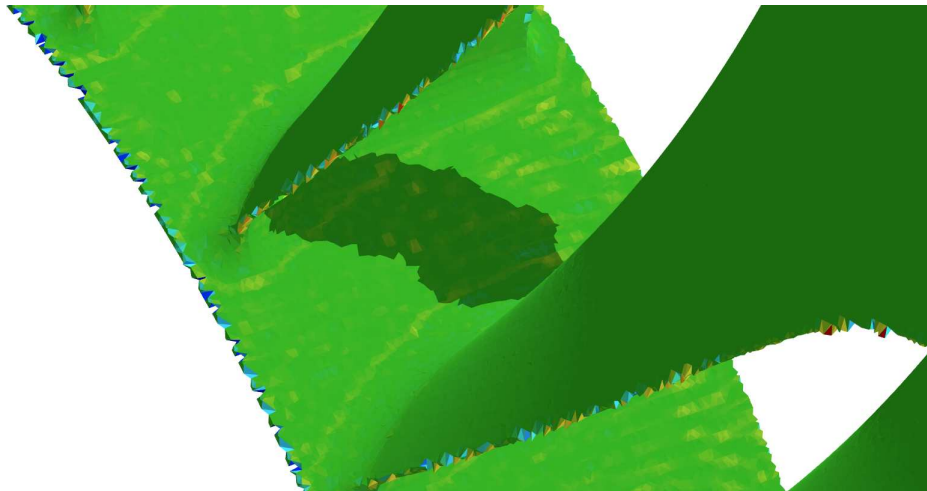


(b) Demo Blisk 0.03 mm.

Source: Author's archive.

An issue becomes apparent in the result for the Demo Blisk model. Figure 37 provides a detailed view of the problematic area from the full deviation map (Figure 36) to better illustrate this discrepancy.

Figure 37 – "Shadow-like" artifact in the demoblisk digital twin generated by Vedo.



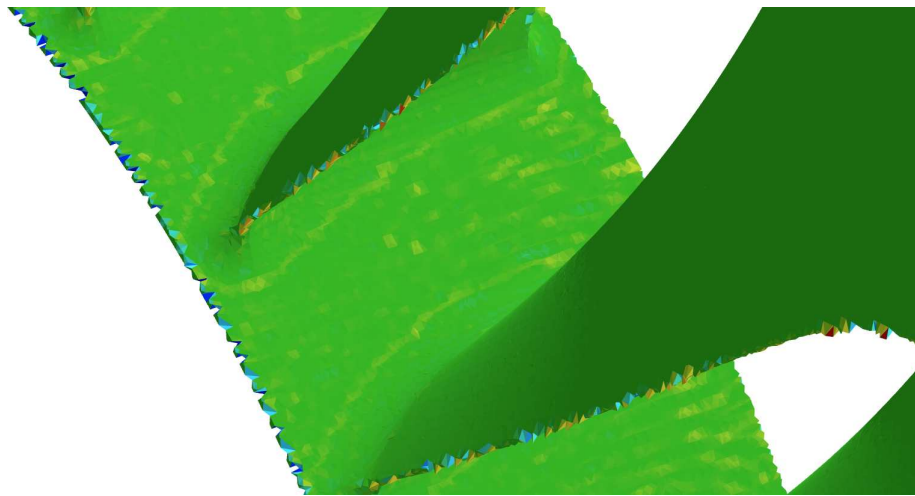
Source: Author's archive.

The visual artifact is due to how Vedo computes the normals of the triangles. Vedo can compute and store smoothed or averaged normals in the mesh's cell data, depending on the mesh topology and its internal settings. In contrast, the PyVista implementation computes per-triangle normals directly from the triangle vertices, which are always flat and perpendicular to the face.

The STL file format stores a single normal per triangle, which 3D viewers use for shading. If a smoothed normal that is not perpendicular to the triangle is written to the file, the viewer renders the lighting incorrectly, creating the shadow artifact. Flat normals, as generated in the PyVista code, do not cause this issue because the shading is uniform for each triangle.

To confirm this was the cause, the method for calculating normals was changed in the Vedo implementation to match the one used for PyVista. The result, shown in Figure 38, no longer contains the shadow artifact.

Figure 38 – Digital Twin generated by new normal calculation process in Vedo.



Source: Author's archive.

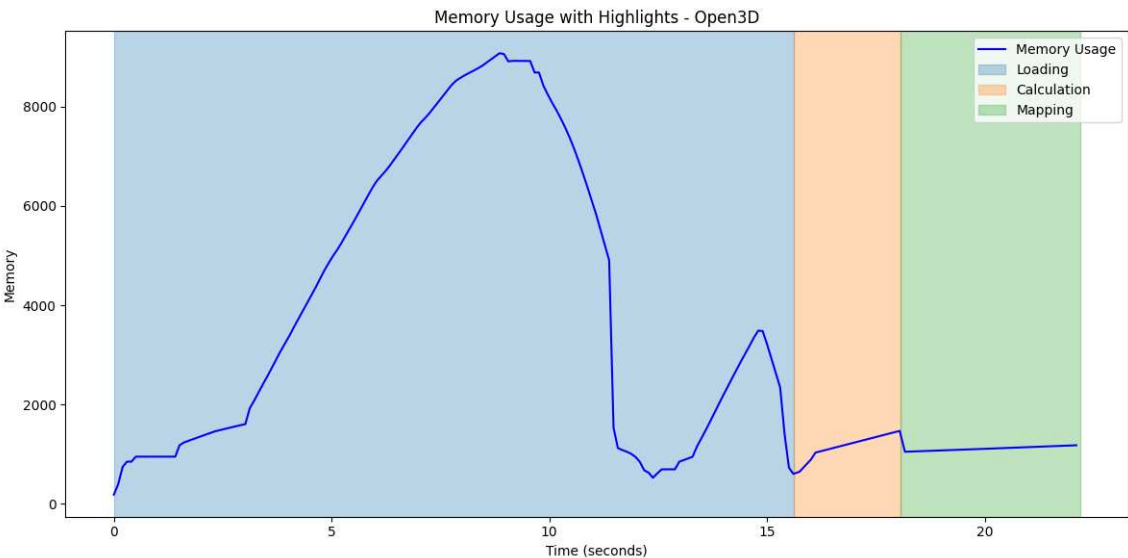
This change to the normal calculation does not significantly affect the implementation's performance. For this reason, the metrics previously presented in Table 6 remain valid and a new performance evaluation is not required.

6.5 TRANSFORMING

During the testing phase, a notable performance discrepancy was observed in the Open3D implementation. It consistently demonstrated longer loading times and higher memory consumption across the overall implementation compared to the other libraries. To pinpoint the source of this overhead, an analysis was performed to measure the time taken by each major section of the code: loading the mesh, computing the distances and exporting the final result.

The results of this test, shown in Figure 39, clearly identify the initial file loading as the primary bottleneck, consuming a disproportionate amount of time and resources. Based on this finding, a hybrid solution was devised. This approach seeks to combine the strengths of multiple libraries by using a more lightweight library, such as PyVista and Vedo, for its faster and less resource-intensive file loading. Once loaded, the mesh data is then transferred to Open3D to take advantage of its highly efficient raycasting for the signed distance calculation, which was identified as the fastest among the tested libraries.

Figure 39 – Open3D Memory Usage segmented by sections.



Source: Author's archive.

Tables 7 and 8 show the results for the two transformation methods:

Table 7 – Transforming metrics (Vedo - Open3D).

Test Object	Avg. CPU (%)	Avg. Max CPU (%)	Max CPU (%)	Avg. Memory (MB)	Avg. Max Memory (MB)	Max Memory (MB)	Duration (s)
Alpha work-piece	8.77	66.28	71.20	902.62	1149.47	1152.91	30.56
Demo Blisk	9.33	82.51	90.04	718.34	1322.91	1327.03	22.28
Jet Bracket	9.43	70.65	75.96	494.93	601.70	604.85	12.58
Test Brick	10.86	82.17	92.24	249.86	272.93	274.98	2.85
PSM Demo.	21.77	51.83	71.67	188.10	194.09	194.59	0.31

Source: Author's archive.

Table 8 – Transforming metrics (Pyvista - Open3D).

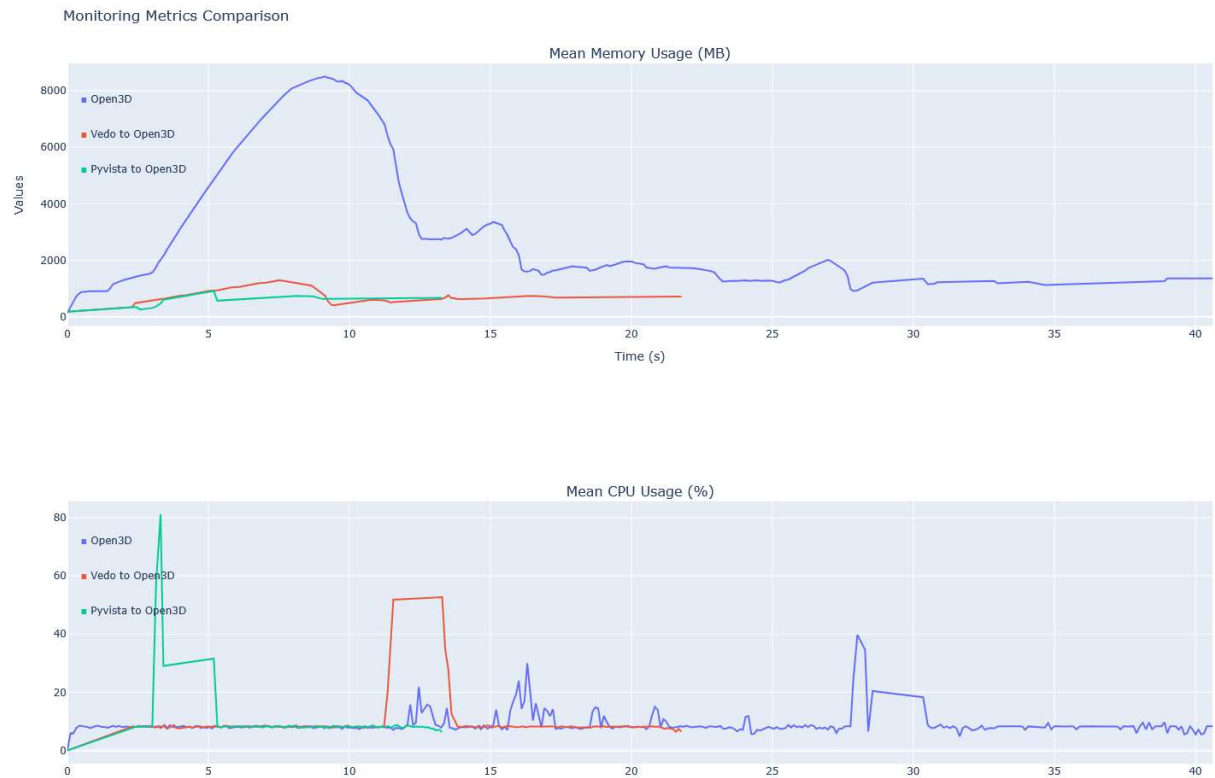
Test Object	Avg. CPU (%)	Avg. Max CPU (%)	Max CPU (%)	Avg. Memory (MB)	Avg. Max Memory (MB)	Max Memory (MB)	Duration (s)
Alpha work-piece	8.88	72.28	79.02	947.43	1149.01	1152.89	22.41
Demo Blisk	10.10	81.18	94.38	629.78	916.02	917.29	13.08
Jet Bracket	9.76	63.46	69.79	506.229	600.93	605.20	9.23
Test Brick	12.07	79.61	85.08	249.82	264.32	267.36	1.96
PSM Demo.	29.78	72.12	84.04	188.50	192.44	193.17	0.18

Source: Author's archive.

An analysis of the tables reveals that this hybrid approach yields a significant reduction in both memory consumption and overall simulation duration. Furthermore, a slight decrease was observed in CPU metrics, including both the maximum and mean percentage usage across the test runs. Figure 40 provides a direct visual comparison, highlighting these performance gains against the standard Open3D implementation.

It is important to note that since the core distance calculation algorithm is identical to the one used in the original Open3D method, the accuracy of the deviation results is unaffected. Consequently, a separate quality assessment of this hybrid approach is unnecessary.

Figure 40 – Monitoring Metrics comparison between Open3D and Transformation methods.



Source: Author's archive.

6.6 TRIMESH

The Trimesh library was also evaluated. It is a widely used Python library for loading and using triangular meshes, with a stated emphasis on providing a well-tested object for easy manipulation of watertight surfaces.

Initial tests with this implementation showed promising results on the small PSM Demonstrator model, where its performance was competitive. However, when applied to the larger test meshes, the implementation did not scale effectively. Execution times became really long and the process frequently led to instability, causing the Integrated Development Environment (IDE) to crash or kill the process due to high resource consumption. Given these significant performance and stability issues on larger datasets, further testing with Trimesh was discontinued. Consequently, no performance metrics or generated Digital Twins for this implementation are presented in this work.

6.7 FRONTEND

As a potential way to streamline the workflow within the gemineers software and product, the implementation of deviation calculation was also carried out in the frontend. The goal is to eliminate the need for a separate service, thus removing the dependency on an external service call. This approach can lead to several benefits, including improved performance by reducing latency, simplified architecture with fewer services to manage and reduced system resource consumption, as there would be no need to run and maintain an additional service. Moreover, by handling the calculation directly in the frontend, a more responsive user experience could be achieved, with faster updates and less reliance on backend operations. If the solution turns out to be significantly less effective than alternative approaches, or if eliminating the service proves unfeasible, it could still be explored in a different way, potentially as a new feature, depending on its performance and results.

The development of this implementation will follow the activity diagram proposed in Figure 23. However, instead of using a standard raycasting technique, a Three.js plugin that uses BVH to optimize spatial queries and raycasting efficiency was used. This approach enhances performance by significantly reducing the number of intersection tests, making the deviation calculation more efficient when handling complex geometries.

To evaluate the efficiency of different BVH methods for computing deviation metrics in the frontend, three approaches were considered: BVH without parallelism, BVH with a single worker for UI responsiveness and BVH with maximum available workers for parallel distance calculation. These methods were benchmarked across geometries of varying complexity, characterized by different numbers of points.

The discussion regarding the development, architecture, and comparative trade-offs of the single-worker and multi-worker implementations has been removed from this work due to confidentiality requirements.

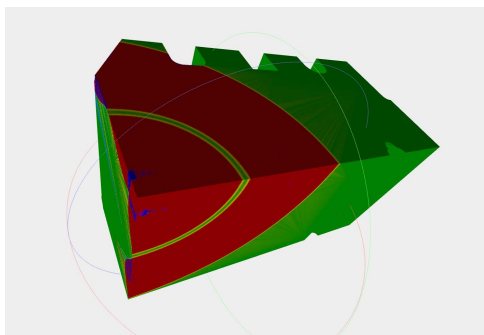
6.7.1 BVH

The three-mesh-bvh plugin for Three.js is used, this plugin generates the Bounding Volume Hierarchy structure, as described in the theoretical background, directly from the mesh's geometry. It then patches the mesh's default raycasting function, replacing the inefficient linear search with a highly optimized traversal of the BVH tree.

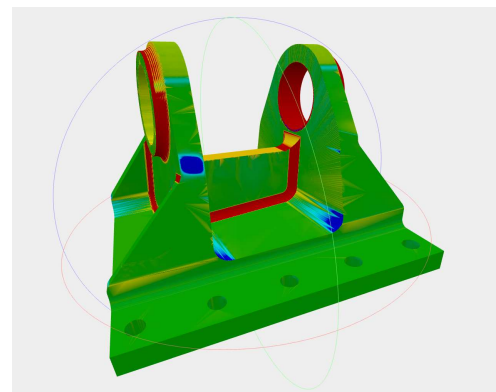
The implementation of this BVH-based calculation is consistent across the different frontend approaches, including those run in Web Workers. The process consists of first generating a BVH for the reference mesh. Then, for each point of the source mesh, the distance magnitude is found by using the BVH to efficiently find the closest point on the reference surface. The sign of this distance is then determined using a dot product calculation, the same way done in the backend implementations. This is performed between two vectors: the normal of the triangle face containing the closest point and the direction vector from that surface point to the source point. The final signed distance is then used to assign a color to the point for visualization.

To validate the results of the frontend implementation, the generated Digital Twins are compared against the established benchmark from the backend method. It is important to note that since the single-worker and multi-worker approaches use the identical underlying BVH calculation logic, they produce visually identical results. Therefore, the output shown in Figure 41 is representative of all three frontend implementations. Although the results are broadly comparable to the backend solutions, the frontend implementation is highly sensitive to the quality of the input mesh. This is because it samples distances from the mesh vertices instead of the triangle centroids, making it more susceptible to local topological errors. This sensitivity is most noticeable in the results for the Test Brick and Alpha workpiece models.

Figure 41 – Digital twins produced by frontend implementation with 0.03 mm deviation.



(a) Test Brick.



(b) Jet Bracket.

Source: Author's archive.

6.8 COMPARATIVE ANALYSIS

This subsection begins the comparative analysis by summarizing the key findings from the different implementations. The performance metrics for the standalone and hybrid approaches are in Tables 4 through 8.

The standalone Open3D implementation is characterized by high resource consumption. Its memory usage is particularly high, peaking at over 9 GB for the Demo Blisk model. This is attributed to the *t.io.read_triangle_mesh* function, which is part of Open3D's modern tensor-based architecture. This design may pre-allocate large memory blocks for computational efficiency, which benefits calculation speed but results in a large memory footprint during the loading phase. High CPU utilization was also observed, with peak usage often exceeding 85%. Notably, the mean CPU usage for the smallest workpiece, the PSM Demonstrator, was 32.25%. This is considerably higher than the mean usage for the larger models, suggesting that the overhead of the process is significant relative to the very short execution time for small files.

In contrast, the VTK-based libraries, PyVista and Vedo, are more memory and cpu efficient, with usage staying below 1.6 GB for bigger workpieces. This efficiency has other trade-offs. The CPU usage for these libraries is much lower than for Open3D, suggesting their core methods are not as parallelized. Qualitatively, the PyVista implementation produced artifacts on the flawed Test Brick mesh, while the Vedo implementation was the most robust in handling this flawed data but had the longest execution times.

The hybrid implementations were created to combine the low-memory loading of the VTK-based libraries with the fast calculation of Open3D. This strategy was effective, both hybrid methods significantly reduced the memory footprint compared to standalone Open3D. Regarding CPU utilization, while the hybrid methods showed an increase compared to the VTK-only approaches, their overall usage was still lower than that of the standalone Open3D implementation. This indicates that Open3D's calculation methods introduce a notable CPU overhead, independent of the resource-intensive loading process. Despite this, the hybrid approaches achieved the lowest overall execution times for the large and complex workpieces. The PyVista-Open3D combination emerged as the top performer, delivering the fastest results across most test cases.

To validate the initial findings and create a more controlled testing environment, a Virtual Machine (VM) was configured. A VM is a software-based emulation of a physical computer that runs on a host machine but operates as a self-contained, isolated system with its own dedicated resources.

The VM was configured with a Linux operating system. This choice was made for several reasons. First, Linux environments are standard for deploying backend services, making this test a closer approximation of a real-world production scenario. Second, many scientific and open-source libraries, including those used in this project, are often

developed and optimized primarily for Linux, which can lead to better stability and performance.

Running the tests within this isolated environment provides more precise and repeatable results. It ensures that the measured CPU and memory consumption are almost exclusively attributable to the implementation being tested, eliminating potential interference from other applications or system activities present on the host machine.

The VM was allocated specifications matching the host machine used for the initial tests: 12 processor cores and 8 GB of RAM. A key distinction, however, is the absence of a dedicated graphics processing unit (GPU). This ensures that all computations are strictly CPU-bound, which is representative of how this service would operate in a server environment without graphical hardware acceleration, gemineers's case. The results for each implementation and the hybrid approaches can be seen in Tables 9 through 13.

Table 9 – VM Open3D metrics.

Test Object	Avg. CPU (%)	Avg. Max CPU (%)	Max CPU (%)	Avg. Memory (MB)	Avg. Max Memory (MB)	Max Memory (MB)	Duration (s)
Alpha work-piece	9.36	36.72	38.83	2886.79	4329.32	4329.54	11.44
Demo Blisk	9.56	25.32	58.29	3714.32	5250.19	5280.79	11.62
Jet Bracket	9.97	43.77	48.49	1307.99	2017.23	2037.69	3.18
Test Brick	15.49	56.81	59.03	485.08	546.27	546.50	0.70
PSM Demo.	0.00	0.00	0.00	335.13	335.13	335.44	0.00

Source: Author's archive.

Table 10 – VM Pyvista metrics.

Test Object	Avg. CPU (%)	Avg. Max CPU (%)	Max CPU (%)	Avg. Memory (MB)	Avg. Max Memory (MB)	Max Memory (MB)	Duration (s)
Alpha work-piece	8.47	39.34	49.51	1262.30	1760.67	1779.61	30.60
Demo Blisk	8.49	12.45	20.04	1145.26	1514.79	1540.91	21.46
Jet Bracket	8.41	16.16	23.25	781.37	979.42	1043.66	14.13
Test Brick	8.42	12.82	27.99	507.81	548.41	549.74	6.51
PSM Demo.	4.04	8.07	9.08	421.08	425.43	427.14	0.11

Source: Author's archive.

Table 11 – VM Vedo metrics.

Test Object	Avg. CPU (%)	Avg. Max CPU (%)	Max CPU (%)	Avg. Memory (MB)	Avg. Max Memory (MB)	Max Memory (MB)	Duration (s)
Alpha work-piece	8.53	48.88	49.88	1085.29	1715.72	1765.35	42.82
Demo Blisk	8.65	49.29	49.92	1202.25	1455.14	1459.49	27.39
Jet Bracket	8.93	48.95	49.04	649.78	986.54	1015.00	14.15
Test Brick	9.01	49.05	49.90	471.47	556.50	563.87	12.03
PSM Demo.	13.43	49.13	49.89	299.64	431.01	436.99	1.47

Source: Author's archive.

Table 12 – VM Transforming metrics (Vedo - Open3D).

Test Object	Avg. CPU (%)	Avg. Max CPU (%)	Max CPU (%)	Avg. Memory (MB)	Avg. Max Memory (MB)	Max Memory (MB)	Duration (s)
Alpha work-piece	8.68	36.09	37.81	895.21	1684.79	1689.91	11.65
Demo Blisk	11.31	45.12	47.82	927.04	1594.02	1601.15	11.38
Jet Bracket	11.35	38.60	39.44	647.39	1048.01	1050.23	4.25
Test Brick	12.55	49.14	60.01	471.35	517.59	523.56	1.15
PSM Demo.	0.00	0.00	0.00	414.16	414.56	414.84	0.00

Source: Author's archive.

Table 13 – VM Transforming metrics (Pyvista - Open3D).

Test Object	Avg. CPU (%)	Avg. Max CPU (%)	Max CPU (%)	Avg. Memory (MB)	Avg. Max Memory (MB)	Max Memory (MB)	Duration (s)
Alpha work-piece	7.43	33.08	40.30	1106.08	1641.99	1642.35	5.89
Demo Blisk	7.81	19.01	19.60	792.64	986.19	988.69	5.20
Jet Bracket	10.09	42.71	44.71	667.45	1035.93	1040.11	1.66
Test Brick	16.74	53.61	56.35	480.21	512.52	542.46	0.41
PSM Demo.	0.00	0.00	0.00	422.66	428.38	429.70	0.00

Source: Author's archive.

Notable changes can be observed in the standalone Open3D implementation (Table 9). Execution times were drastically reduced across all workpieces. While memory consumption also decreased compared to the initial tests, it remains significantly higher than the VTK-based alternatives. Peak CPU utilization was also substantially lower, reduced by nearly 50% in most cases. It is important to note that the metrics for the PSM Demonstrator case in this test are unreliable. With a mean execution time of only 0.004 seconds, the overhead from the resource monitoring process itself likely dominates the measurement, making the recorded CPU and duration values inaccurate.

The VTK-based libraries, PyVista (Table 11) and Vedo (Table 10), also saw improvements in execution speed within the Linux VM environment, particularly on the larger workpieces. However, this came with a general increase in both memory and CPU consumption compared to the initial tests. This suggests that while the core algorithms may be more performant on Linux, the resource management or process overhead within the VM environment is different, leading to a higher baseline consumption.

The hybrid approaches demonstrated the most significant performance gains in the controlled VM environment. Execution times were reduced dramatically, with the PyVista-Open3D implementation (Table 13) processing the largest workpieces in approximately 5-6 seconds. This speed was achieved with a lower peak CPU utilization compared to the initial non-VM tests. Memory usage remained comparable to the initial tests, showing a slight increase consistent with the behavior of the VTK-based loaders in the VM.

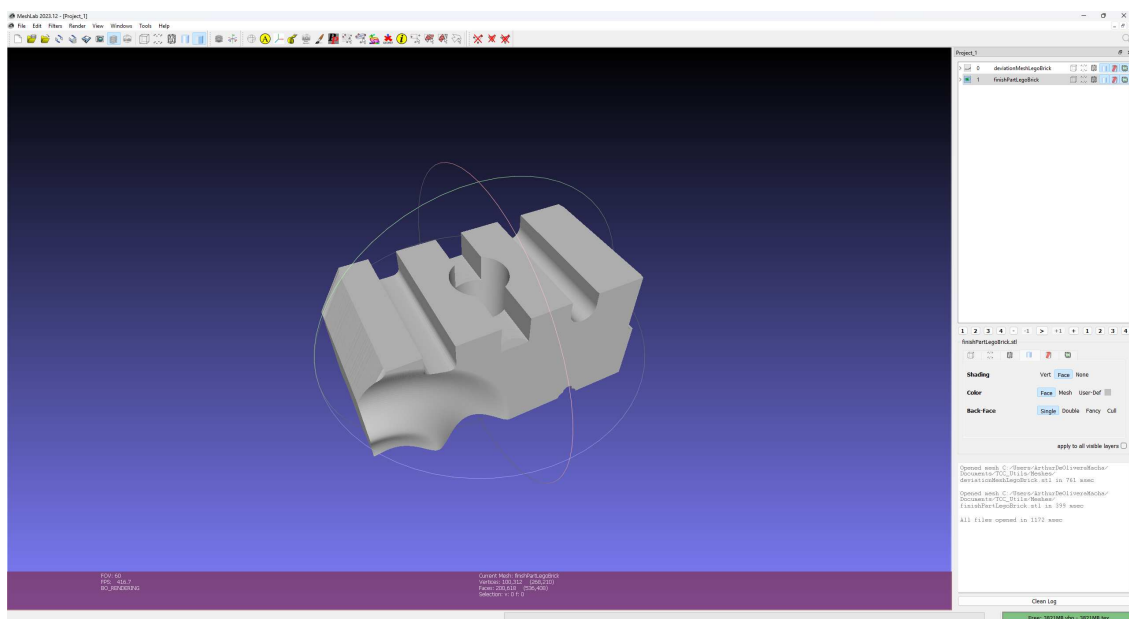
6.9 BENCHMARKING WITH NON-COMPETITOR COMPANIES

Deviation calculation with MeshLab

To benchmark the developed solution against an industry standard for both accuracy and performance, the open-source software MeshLab was selected. MeshLab is a system for processing and editing 3D triangular meshes, offering a comprehensive set of tools for editing, cleaning, healing, inspecting and rendering. It is frequently used to process raw data from 3D digitization devices and to prepare models for 3D printing. The software's user interface is shown in Figure 42.

Due to its nature as a desktop application, the performance comparison is limited to the total execution time, as a detailed resource analysis of CPU and memory is not feasible.

Figure 42 – MeshLab Interface.



Source: Author's archive.

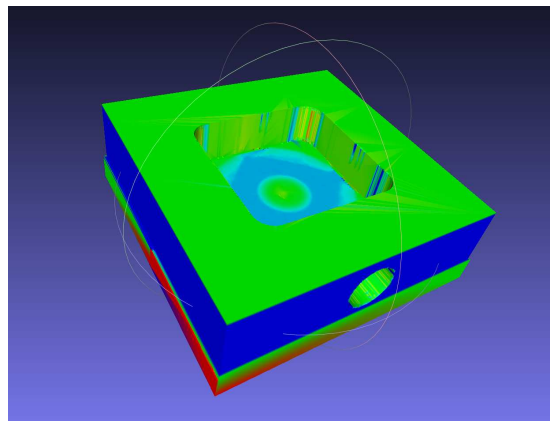
For the signed distance calculation in MeshLab, the "Distance from Reference Mesh" function is used. This function operates on the principle of surface sampling, taking points from one mesh (typically its vertices) and finding the closest corresponding point on the surface of the reference mesh. To perform this search efficiently, MeshLab's underlying VCGlib library employs a spatial data structure, such as a Uniform Grid or an Axis-Aligned Bounding Box (AABB) tree, which accelerates the process by quickly discarding distant regions.

Once the closest point on the surface is found, the sign of the distance is determined by comparing the direction vector (from the surface point to the sample point)

with the surface normal at that location, using a dot product. It is noteworthy that this method for assigning the sign is conceptually identical to the approach developed for the implementations in this work. This alignment with an industry-standard tool serves as a validation of the correctness of the methodology used in this thesis.

The final signed distance values are then stored as a per-vertex 'quality' attribute on the source mesh, which can be visualized through a color map. The mapping for the PSM Demo. workpiece is shown in Figure 43.

Figure 43 – Color Mapping Result for PSM Demo.



Source: Author's archive.

An initial analysis of the PSM Demonstrator result reveals that while the color map used by MeshLab differs from the one in this project, the conceptual representation of deviation is consistent. However, a notable artifact is present on the underside of the workpiece, where an incorrect deviation is displayed. This error is attributed to the mesh not being fully manifold in that region, indicating that MeshLab's accuracy is also dependent on the topological integrity of the input data.

A quantitative comparison of the signed distance statistics further reinforces the validity of the developed implementations. The key metrics are as follows in Table 14.

Table 14 – Signed distances metrics for PSM Demo. (MeshLab).

Library	Min	Max	Mean	RMS
MeshLab	-1.52	0.50	-0.1056	0.3498
Open3D/Hybrid	-1.608	0.50	-0.1042	0.3480
PyVista	-1.608	0.50	-0.1042	0.3480
Vedo	-1.510	0.50	-0.1049	0.3430

Source: Author's archive.

This high degree of statistical correlation is significant. The maximum deviation is identical across all methods and the mean and Root Mean Square (RMS) values are

nearly the same, indicating that the overall magnitude and distribution of calculated deviations are consistent. The minor variations in the minimum value are likely attributable to small differences in how each library's algorithm samples the surface or resolves the closest point in complex regions.

Interestingly, this specific artifact did not appear in any of the implementations developed in this thesis, suggesting they may be more robust in handling this particular type of mesh flaw. In terms of performance, the calculation in MeshLab took approximately 40 milliseconds. This execution time is comparable to the fastest methods tested in this work, namely the standalone Open3D and the hybrid implementations.

To determine if these findings scale to larger models, the analysis was repeated for the Alpha workpiece.

The visual artifacts observed in the MeshLab output are also present on the larger Alpha workpiece model. This finding further indicates that the implementation's accuracy is dependent on the topological quality of the input mesh, not its scale.

To validate this conclusion, a two-step mesh conditioning process was performed using MeshLab's built-in tools before re-running the distance calculation.

- **Repair Non-Manifold Edges:** The first step used a filter to repair non-manifold edges by splitting vertices. This tool identifies vertices where more than two distinct surfaces are improperly joined (a non-manifold condition). It resolves this by duplicating the single vertex, creating separate, correctly connected vertices for each continuous surface, thereby making the mesh manifold.
- **Midpoint Subdivision:** After the repair, a midpoint subdivision scheme was applied. This process uniformly refines the mesh by splitting every edge at its midpoint, which replaces each original triangle with four smaller ones. This increases the mesh resolution and creates a better-defined surface for calculation.

With the mesh now topologically correct and more finely tessellated, the distance calculation was performed again. It was expected that this would produce a more accurate deviation map, which would consequently alter the statistical results and the measured execution time.

Once the mesh is made manifold through the conditioning process, the resulting deviation map is correctly colored and free from the previous artifacts. This correction, however, required an increase in mesh complexity, raising the vertex count from 1,775,772 to 2,139,137 and the face count from 3,550,349 to 4,274,781.

The mean execution time for the distance calculation alone on this refined mesh was 3.642 seconds. However, this measurement does not account for the initial loading of the workpiece, which took an average of 12 seconds. This brings the total workflow time in MeshLab to over 15 seconds for this model. This complete execution time is considerably longer than that of the hybrid implementations developed and almost the

same time as the Open3D, but still significantly faster than the standalone VTK-based approaches.

For a complete statistical comparison, the analysis will now be presented for both the original and the repaired mesh. This dual approach allows for an evaluation of how the algorithms perform on flawed input data, followed by a baseline comparison on clean, manifold geometry.

A statistical comparison of the signed distance values for the original Alpha workpiece mesh provides further quantitative insight. The key metrics for each implementation are in Table 15:

Table 15 – Signed distances metrics for Alpha workpiece (MeshLab).

Library	Min	Max	Mean	RMS
MeshLab	-8.17	8.84	0.3744	1.2403
Open3D/Hybrid	-8.53	8.95	0.3755	1.2400
PyVista	-8.53	8.95	0.3755	1.2400
Vedo	-3.70	8.68	0.3760	1.2387

Source: Author's archive.

This data reveals a high degree of consistency across most metrics. The mean deviation is calculated at approximately 0.375 by all four methods and the Root Mean Square (RMS) values are nearly identical around 1.24. This indicates that the overall magnitude and statistical distribution of the calculated deviations are similar. The maximum positive deviations are also within a very close range.

The most notable discrepancy is found in the minimum deviation value calculated by Vedo. While PyVista, Open3D and MeshLab report minimum values in the range of -8.17 to -8.53, Vedo reports a significantly less extreme value of -3.70. This suggests that Vedo's *vtkDistancePolyDataFilter* may handle certain topological artifacts or extreme local errors differently than the other methods. It appears to be less sensitive to the specific mesh flaws that produce the most extreme negative deviation values, which aligns with the previous qualitative observation of its robustness in generating visually clean results on imperfect meshes.

Following the mesh conditioning process, a new statistical analysis was performed on the now-manifold Alpha workpiece model. The results reveal a convergence in the overall deviation profile across all implementations (Table 16).

Table 16 – Signed distances metrics for new Alpha workpiece manifold (MeshLab).

Library	Min	Max	Mean	RMS
MeshLab	-6.81	9.56	0.2904	1.0798
Open3D/Hybrid	-3.72	9.24	0.2882	1.0766
PyVista	-7.82	9.24	0.2881	1.0766
Vedo	-3.57	9.23	0.2890	1.0721

Source: Author's archive.

The mean and Root Mean Square (RMS) values are now extremely close, with all methods calculating a mean deviation of approximately 0.29 and an RMS of approximately 1.07. This strong statistical agreement on the corrected mesh confirms that all tested methods produce a consistent and reliable overall result when provided with high-quality input data.

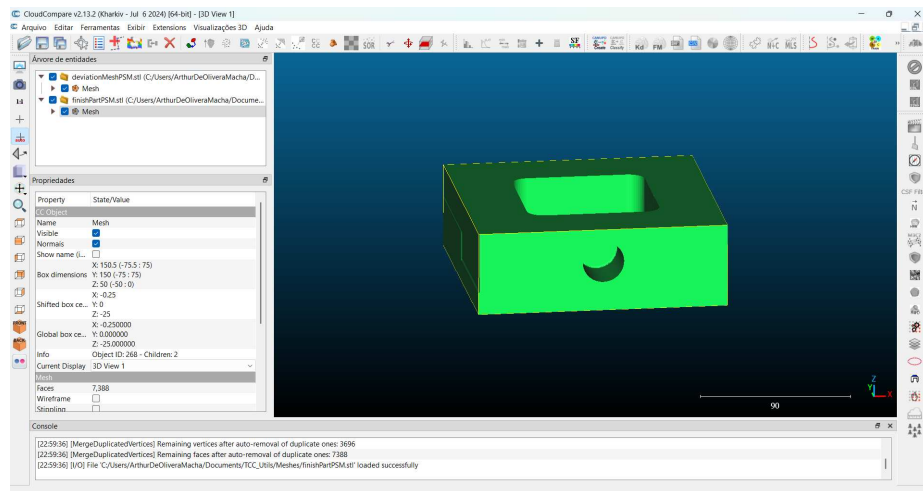
While the central tendency and distribution are consistent, the extreme values still show minor but informative variations. The maximum positive deviation is correlated, with only MeshLab reporting a slightly higher value. The most significant divergence remains in the minimum deviation values. On the corrected mesh, Open3D and Vedo report similar, less extreme minimums (-3.72 and -3.57, respectively), while PyVista and MeshLab report more extreme negative values (-7.82 and -6.81).

In conclusion, this detailed statistical comparison validates the open-source implementations developed in this thesis. The strong correlation of the mean and Root Mean Square (RMS) values, particularly on the clean, manifold mesh, demonstrates that all methods produce a statistically consistent and reliable overall result when provided with high-quality input data. However, the analysis also reveals key distinctions. The persistent variations in the minimum deviation values, both on the original and repaired meshes, expose fundamental differences in how each algorithm handles local geometric complexities and mesh artifacts. Vedo's robustness to certain flaws on the original mesh and the distinct behaviors of PyVista and Open3D on the repaired mesh highlight that no single method is universally superior. Ultimately, this quantitative analysis confirms that the developed open-source solutions are not only performant but also produce deviation results that are statistically comparable to the established MeshLab standard, confirming their suitability as effective replacements.

Deviation calculation with CloudCompare

The final benchmark compares the developed solution against CloudCompare, an open-source software for processing 3D point clouds and meshes that now supports direct mesh-to-mesh distance calculation. As a desktop application, the performance comparison is limited to execution time and the analysis will also include a statistical comparison of the signed distance values. The interface for CloudCompare is shown in Figure 44.

Figure 44 – CloudCompare Interface.

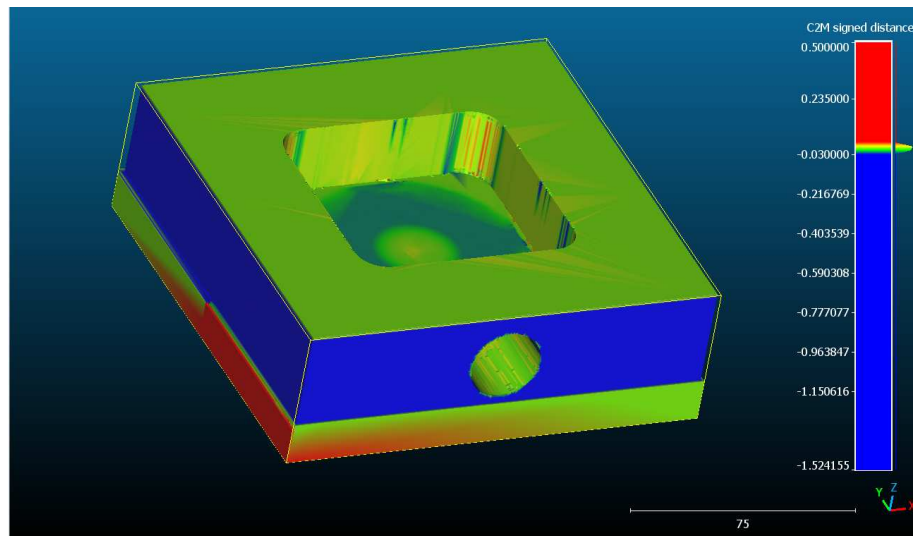


Source: Author's archive.

The "Cloud/Mesh Distance" (C2M) tool in CloudCompare calculates distance by first finding the closest triangle on the reference mesh for each point of the source cloud or mesh. The calculation follows a two-case logic: it first checks the perpendicular distance to the triangle's face, but if the point's projection falls outside the face, it then computes the distance to the nearest edge to ensure the true minimum is found. To make this search efficient, CloudCompare employs an Octree as a spatial index. The sign of the distance is determined by a dot product between the normal of the closest triangle and the direction vector from the surface point to the query point. The same mechanism used by MeshLab and the manual dot product method developed for the PyVista implementation in this thesis.

The comparison is now performed on the PSM Demonstrator and Alpha work-piece models. The resulting deviation maps, as calculated by this algorithm, are visualized in Figure 45.

Figure 45 – Color mapping for PSM Demo. at 0.03 mm.



Source: Author's archive.

The initial observation of the CloudCompare output is a different color map, which is a cosmetic distinction and not relevant to the accuracy analysis. More significantly, the same visual artifacts seen in the MeshLab results are present here and are notably more pronounced, even on the PSM Demonstrator model. This suggests that CloudCompare's algorithm is exceptionally sensitive to the topological integrity of the input data, even more so than the other tools evaluated.

This finding also highlights the robustness of the centroid-based sampling method used in this thesis. By querying distances from the center of each triangle rather than its vertices, the implementation is less susceptible to localized errors at non-manifold edges or improperly connected vertices.

Regarding performance, the calculation for the PSM model took an average of 53.16 seconds. It is important to note that this execution time is influenced by the "Octree level" parameter. While CloudCompare suggests an optimal level, this setting can be manually adjusted and the resulting performance at any given level is dependent on the available hardware resources.

Table 17 – Signed distances metrics for PSM Demo. (CloudCompare).

Library	Min	Max	Mean	RMS
CloudCompare	-1.52	0.50	-0.1055	0.3335
Open3D/Hybrid	-1.608	0.50	-0.1042	0.3480
PyVista	-1.608	0.50	-0.1042	0.3480
Vedo	-1.510	0.50	-0.1049	0.3430

Source: Author's archive.

A statistical analysis of the PSM Demonstrator results reveals that the minimum, mean and RMS values align closely with those from the developed implementations, while the maximum deviation value is identical.

The execution time for the Alpha workpiece model was approximately 644 seconds.

Table 18 – Signed distances metrics for Alpha workpiece. (CloudCompare).

Library	Min	Max	Mean	RMS
CloudCompare	-3.78	8.84	0.3760	1.1819
Open3D/Hybrid	-8.53	8.95	0.3755	1.2400
PyVista	-8.53	8.95	0.3755	1.2400
Vedo	-3.70	8.68	0.3760	1.2387

Source: Author's archive.

In the statistical results, the minimum deviation value calculated by CloudCompare is very close to that of the Vedo implementation. However, it is significantly less extreme than the minimums reported by MeshLab, Open3D and PyVista.

While the mean, max and RMS values remain consistent across all methods, this divergence in the minimum value suggests that for complex geometries, the algorithms used by Open3D and PyVista are capable of identifying more extreme negative deviations, providing more detailed information in areas of deep concavity or complex features.

7 CONCLUSION AND FUTURE WORK

This project was initiated to address performance bottlenecks in the proprietary software used by gemineers GmbH for quality control in machining processes. The primary objective was to benchmark open-source libraries and develop a high-performance alternative for 3D object comparison, capable of handling the complex geometries typical of components like blisks without compromising speed or accuracy.

To achieve this primary goal, the specific objectives outlined at the beginning of this work were successfully met. This was accomplished by establishing a standardized workflow for deviation analysis, which was then implemented and benchmarked using various open-source tools. This general workflow consists of loading the reference and simulated part geometries, calculating the shortest distance from the centroids of each triangle in the simulated part to the reference surface, and determining the sign of this distance (inside or outside) by performing a dot product with the surface normal. The investigation revealed that a hybrid implementation of this workflow, which leverages the distinct advantages of multiple libraries, yielded the most performant solution. The final process involves using PyVista for its fast and memory-efficient loading of STL files, after which the mesh data is transferred to Open3D to utilize its raycasting-based algorithm for the signed distance calculation.

The results demonstrated that this hybrid solution outperformed both the proprietary software and the standalone open-source configurations. This method yielded a reduction in both execution time and memory consumption, particularly when tested in a controlled Linux-based virtual machine environment that simulates a production server. The chosen methodology, combining the agile Scrum framework with a structured benchmarking process, proved crucial for validating these results across multiple scenarios. Furthermore, the accuracy of the developed solution was confirmed by benchmarking it against established industry tools like MeshLab and CloudCompare. The statistical analysis of the deviation results showed a high degree of correlation, confirming that the open-source implementation produces reliable and accurate results consistent with industry standards. The findings validate that a well-architected open-source solution can effectively replace the existing proprietary tool, offering gemineers GmbH a path to increase the efficiency of its digital twin assessment platform and gain greater architectural flexibility while also reducing costs.

Based on the findings of this research, several avenues for future work have been identified that could further enhance the performance, robustness, and functionality of the developed solution.

A primary area for future investigation is the enhancement of computational performance through hardware acceleration. The Open3D library offers support for GPU acceleration via CUDA, and a significant next step would be to explore the per-

formance impact of offloading the loading of files and distance calculations to a GPU. For environments equipped with capable graphics hardware, this could lead to another order-of-magnitude reduction in computation time, making near-real-time analysis of very large and complex meshes feasible. Additionally, to improve the robustness of the system, future work should focus on integrating automated mesh conditioning steps into the workflow. This research consistently highlighted the sensitivity of deviation algorithms to the topological quality of the input mesh. By using library functions to detect and repair common issues such as non-manifold edges, T-junctions, and inverted normals prior to the deviation calculation, the entire process would become more resilient and less reliant on the quality of upstream data.

Further development could also focus on expanding the tool's functionality and user interaction capabilities. While the backend solution proved to be performant, the frontend implementation using three-mesh-bvh did not scale as effectively for large models. However, this approach could still be repurposed for specific use cases, such as providing a rapid, low-quality preview for smaller geometries or for decimated versions of larger meshes. This would offer users an immediate, less precise, visualization of the deviation. For a more powerful and fully interactive frontend tool, future research could explore the use of WebGPU to leverage client-side GPU capabilities directly within the browser.

REFERENCES

- ADOBE. **What is ray casting?** Available from: <https://www.adobe.com/products/substance3d/discover/what-is-ray-casting.html>. Visited on: 20 Mar. 2025.
- ANDERSEN, B.; PETTERSEN, P.G. **Benchmarking Handbook**. [S. l.]: Springer Netherlands, 1995. ISBN 9780412735202. Available from: <https://books.google.de/books?id=TU27eYfwVYQC>.
- ANDREWS, Gregory R.; SCHNEIDER, Fred B. Concepts and Notations for Concurrent Programming. **ACM Comput. Surv.**, Association for Computing Machinery, New York, NY, USA, v. 15, n. 1, p. 3–43, Mar. 1983. ISSN 0360-0300. DOI: 10.1145/356901.356903. Available from: <https://doi.org/10.1145/356901.356903>.
- BHUTTA, M.; HUQ, Faizul. Benchmarking – Best Practices: An Integrated Approach. **Benchmarking: An International Journal**, v. 6, p. 254–268, Sept. 1999. DOI: 10.1108/14635779910289261.
- BORRMANN, André; KÖNIG, Markus; KOCH, Christian; BEETZ, Jakob. **Building Information Modeling**. [S. l.]: Springer Cham, 2018. chap. 2. ISBN 978-3-319-92862-3. DOI: 10.1007/978-3-319-92862-3.
- BORZECKI, Maciej; SKURSKI, Adam; KAMINSKI, Marek; NAPIERALSKI, Andrzej; KASPRZAK, Jarosław D.; LIPIEC, Piotr. Applications of Ray-Casting in Medical Imaging. *In*: available from: <https://api.semanticscholar.org/CorpusID:59633860>.
- DE-CAMBRAY, Beatrix; DELPY, Thomas; ZEITOUNI, Karine. A Survey on 3D Data Models in Architecture and Civil Engineering, Dec. 1995.
- CAMP, R.C. **Benchmarking: The Search for Industry Best Practices that Lead to Superior Performance**. [S. l.]: Quality Press, 1989. ISBN 9780527916350. Available from: <https://books.google.de/books?id=PwJPAAMAAMAJ>.
- CHONG, Clara. **Deep Dive into Multithreading, Multiprocessing, and Asyncio**. Available from: <https://towardsdatascience.com/deep-dive-into-multithreading-multiprocessing-and-asyncio-94fdbe0c91f0/>. Visited on: 24 Mar. 2025.
- COOKE, Neil. **How Onshape Has Fundamentally Improved Parametric CAD**. Available from: <https://www.onshape.com/en/blog/how-onshape-has-fundamentally-improved-parametric-cad>. Visited on: 19 Mar. 2025.
- DILBA, Denis. **Blisk development: How blade and disk became one**. Available from: <https://aeroreport.de/en/aviation/blisk-development-how-blade-and-disk-became-one>. Visited on: 2 Dec. 2024.

DOYLE, Michael J.; TUOHY, Ciaran; MANZKE, Michael. Evaluation of a BVH Construction Accelerator Architecture for High-Quality Visualization. **IEEE Transactions on Multi-Scale Computing Systems**, IEEE Computer Society, Los Alamitos, CA, USA, v. 4, n. 01, p. 83–94, Jan. 2018. ISSN 2332-7766. DOI: 10.1109/TMSCS.2017.2695338. Available from: <https://doi.ieeecomputersociety.org/10.1109/TMSCS.2017.2695338>.

DUNHAM, Darin T.; WEST, Philip D. Benchmarks for target tracking. *In: OPTICAL Engineering + Applications*. [S. l.: s. n.], 2011. Available from: <https://api.semanticscholar.org/CorpusID:121634479>.

ELPERN, Jeff; DASCALU, Sergiu M. A Framework for Understanding the Open Source Revolution. **Int. J. Open Source Softw. Process.**, v. 1, p. 1–16, 2009. Available from: <https://api.semanticscholar.org/CorpusID:18363308>.

FOLEY, James D.; DAM, Andries van; FEINER, Steven K.; HUGHES, John F. **Computer graphics: principles and practice (2nd ed.)** USA: Addison-Wesley Longman Publishing Co., Inc., 1990. ISBN 0201121107.

FRAUNHOFER, IPT. **High Performance Cutting: Modern Machining in the Digital Age**. Available from: <https://www.ipt.fraunhofer.de/en/technologies/machining-cutting/high-performance-cutting.html>. Visited on: 7 Jan. 2025.

FU, Feng. Chapter Six - Design and Analysis of Complex Structures. *In: FU, Feng (ed.). Design and Analysis of Tall and Complex Structures*. [S. l.]: Butterworth-Heinemann, 2018. p. 177–211. ISBN 978-0-08-101018-1. DOI: <https://doi.org/10.1016/B978-0-08-101018-1.00006-X>. Available from: <https://www.sciencedirect.com/science/article/pii/B978008101018100006X>.

IANCU, Cătălin; IANCU, Daniel; STĂNCIOIU, Alin. FROM CAD MODEL TO 3D PRINT VIA “STL” FILE FORMAT. **Fiabilitate și Durabilitate**, v. 1, p. 73–80, 2010. Available from: <https://api.semanticscholar.org/CorpusID:55107044>.

KAUFMANN, Jonas. **An Evaluation of Different Measurement Approaches for Feature Performance Analysis**. 2022. MA thesis – Saarland University.

KHRONOS. **glTF 2.0 API Reference Guide**. Available from: <https://www.khronos.org/files/glTF20-reference-guide.pdf>. Visited on: 19 Mar. 2025.

LARSEN, Eric; GOTTSCHALK, Stefan; LIN, Ming; MANOCHA, Dinesh. Fast Proximity Queries with Swept Sphere Volumes, Dec. 1999.

LJUNG, Patric. Efficient Methods for Direct Volume Rendering of Large Data Sets, 2006.

MONTEZ, Carlos. **Concorrência e Tempo Real - Conceitos Básicos de Concorrência e Threads**. [S. l.: s. n.], 2024. Lecture notes from Concorrência e Tempo Real, UFSC, Florianópolis, 2024.

NGUYEN, Sinh Van; TRAN, Ha Manh; MALESZKA, Marcin. Geometric modeling: Background for processing the 3d objects. **Applied Intelligence**, v. 51, p. 6182–6201, 2021. DOI: 10.1007/s10489-020-02022-6.

O'REILLY, Tim. Lessons from open-source software development. **Commun. ACM**, Association for Computing Machinery, New York, NY, USA, v. 42, n. 4, p. 33–37, Apr. 1999. ISSN 0001-0782. DOI: 10.1145/299157.299164. Available from: <https://doi.org/10.1145/299157.299164>.

OSHER, S.; FEDKIW, Ronald. Signed Distance Functions. *In*: available from: <https://api.semanticscholar.org/CorpusID:117338656>.

PENEV, Alexander. Computer graphics and geometric modelling - A hybrid approach. **International Journal of Pure and Applied Mathematics**, v. 85, p. 781–811, July 2013. DOI: 10.12732/ijpam.v85i4.14.

PRASAD, Divya; REDDY, Dr. Ch. Satyananda. Understanding the Differences between Proprietary & Free and Open Source Software. *In*: available from: <https://api.semanticscholar.org/CorpusID:3760197>.

PYVISTA. **Extract Cell Centers**. Available from: <https://docs.pyvista.org/examples/01-filter/cell-centers>. Visited on: 2 Apr. 2025.

ROTH, Scott D. Ray casting for modeling solids. **Computer Graphics and Image Processing**, v. 18, n. 2, p. 109–144, 1982. ISSN 0146-664X. DOI: [https://doi.org/10.1016/0146-664X\(82\)90169-1](https://doi.org/10.1016/0146-664X(82)90169-1). Available from: <https://www.sciencedirect.com/science/article/pii/0146664X82901691>.

SILBERSCHATZ, Abraham; GALVIN, Peter Baer; GAGNE, Greg. **Operating System Concepts**. USA: John Wiley and Sons, Inc., 2012. chap. 4. ISBN 978-1-118-06333-0.

SIM, Susan Elliott; EASTERBROOK, Steve M.; HOLT, Richard C. Using benchmarking to advance research: a challenge to software engineering. **25th International Conference on Software Engineering, 2003. Proceedings.**, p. 74–83, 2003. Available from: <https://api.semanticscholar.org/CorpusID:6036545>.

SONG, Yelin; MA, Winston Wai Shing; yu he yu; ZHOU, Mingdong; WANG, Michael. Stress-based shape and topology optimization with cellular level set in B-splines. **Structural and Multidisciplinary Optimization**, v. 62, Nov. 2020. DOI: 10.1007/s00158-020-02610-7.

SUTHERLAND, Jeff. **The Scrum Handbook**. Available from: <https://www.scruminc.com/wp-content/uploads/2014/07/The-Scrum-Handbook.pdf>. Visited on: 6 Jan. 2025.

SYSTÈMES, Dassault. **What is wireframe modeling and why is it so crucial to design?** Available from: <https://www.3ds.com/store/cad/wireframe-modeling1>. Visited on: 19 Mar. 2025.

TESCHNER, Matthias. **Computer Graphics Ray Casting**. Available from: https://cg.informatik.uni-freiburg.de/course_notes/graphics_01_raycasting.pdf. Visited on: 20 Mar. 2025.

TONG, Jason G.; KHALID, Mohammed A. S. Profiling CAD tools: A proposed classification. **2007 International Conference on Microelectronics**, p. 253–256, 2007. Available from: <https://api.semanticscholar.org/CorpusID:11207052>.

TURK, Greg. **The PLY Polygon File Format**. Available from: <https://gamma.cs.unc.edu/POWERPLANT/papers/ply.pdf>. Visited on: 19 Mar. 2025.

WONG, Kay Jan. **Python Multithreading vs. Multiprocessing Explained**. Available from: <https://builtin.com/data-science/multithreading-multiprocessing>. Visited on: 24 Mar. 2025.

WOODWARD, John R. Generating wireframes from set-theoretic solid models by spatial division. **Computer-aided Design**, v. 18, p. 307–315, 1986. Available from: <https://api.semanticscholar.org/CorpusID:119910766>.

ZEISS. **Blisk: Engineered for performance**. Available from: <https://www.zeiss.de/messtechnik/industrien/aerospace/aircraft-engine/blisk.html>. Visited on: 2 Dec. 2024.

ZHANG, Qi-Hua; SU, Hai-Dong; LIN, Shao-Zhong; SHI, Gen-Hua. Algorithm for three-dimensional curved block cutting analysis in solid modeling. **Computer Methods in Applied Mechanics and Engineering**, v. 360, p. 112721, Nov. 2019. DOI: 10.1016/j.cma.2019.112721.

ANNEX A – CODES USED AS EXAMPLES IN THE PROJECT

Code Snippet A.1 – PyVista documentation example of raytracing.

```

1  h0n = h0.compute_normals(point_normals=True, cell_normals=False,
2  auto_orient_normals=True)
3
4  h0n['distances'] = np.empty(h0n.n_points)
5  for i in range(h0n.n_points):
6      p = h0n.points[i]
7      vec = h0n['Normals'][i] * h0n.length
8      p0 = p - vec
9      p1 = p + vec
10     ip, ic = h1.ray_trace(p0, p1, first_point=True)
11     dist = np.sqrt(np.sum((ip - p) ** 2))
12     h0n['distances'][i] = dist
13
14 # Replace zeros with nans
15 mask = h0n['distances'] == 0
16 h0n['distances'][mask] = np.nan
17 np.nanmean(h0n['distances'])

```