Politecnico di Torino

Politecnico di Torino

Master's degree in Computer Engineering

A.a. 2024/2025

Graduation Session 06 2025

Development of VR web-based solutions to support aerospace simulations

Supervisors:

Andrea Sanna

Eugenio Topa

Author:

Sante Laera

Abstract

3D simulations play a crucial role in the aerospace domain, both for visualizing telemetry data and for enabling interactive operations based on acquired sensor data. These simulations provide intuitive representations that support system monitoring, anomaly detection, and structural analysis during mission planning and execution.

This thesis, carried out at ALTEC, focuses on the development of a TypeScript library and two web-based tools designed to support 3D data visualization and reconstruction of simulated aerospace scenarios on web pages.

The TypeScript library enables the seamless integration of Unity WebGL builds into Vue.js applications. It manages the Unity instance lifecycle and establishes a bidirectional communication layer between the simulation environment and the front-end. This facilitates the integration of interactive visualization within a web interface.

The first tool is a benchmarking framework for evaluating surface reconstruction algorithms applied to LiDAR-generated point clouds. Based on an existing library, it includes a synthetic data generation pipeline using a virtual LiDAR simulator and a dataset of CAD models representing real satellite structures. The framework enables the systematic comparison of remeshing techniques and supports the optimization of algorithm parameters to improve reconstruction quality. The final goal of the tool is to provide ALTEC with an optimized remeshing algorithm.

The second tool is a visualization system for detecting geometric discrepancies between scanned meshes and their reference CAD models. By computing deviations and rendering them as a color-coded heatmap on the target surface, the system highlights potential structural anomalies such as missing components or deformations.

Together, these components form a complete pipeline for data processing and visualization. A demonstration scenario simulating a satellite scanning operation was developed to validate the tools and illustrate their integration in a realistic aerospace context.

Table of Contents

List of Figures

Chapter 1

Introduction

1.1 Context

Three dimensional (3D) simulations have become indispensable in the aerospace sector, offering valuable insights for the design, testing, and execution of missions.

One application of simulation in aerospace mission is to improve monitoring and assessment of operational activities in ground control systems, an example of such simulation is shown in Figure 1.1.

An example of this is the visualization of LiDAR information for object detection and structural assessment. A potential scenario involves a satellite moving along a specific orbital trajectory around another satellite or structure while performing a LiDAR scan, with the objective of identifying defects or missing components. To achieve this, a simulation that, after reconstructing the point cloud, can effectively visualize any structural issues is needed. This process involves integrating three main components: a tool for incorporating the simulation into a ground control system, a mechanism for point cloud reconstruction, and a system to highlight structural problems. Despite the apparent specificity, these tools can be adapted and integrated into a wide range of aerospace missions.

The project was carried out at ALTEC S.p.A., a company that provides engineering and logistics support for activities related to the International Space Station and for the development and execution of planetary exploration missions and other aerospace operations. The development of these tools is therefore aimed at providing ALTEC with solutions that will facilitate the development of various projects that need these simulation and visualization capabilities.

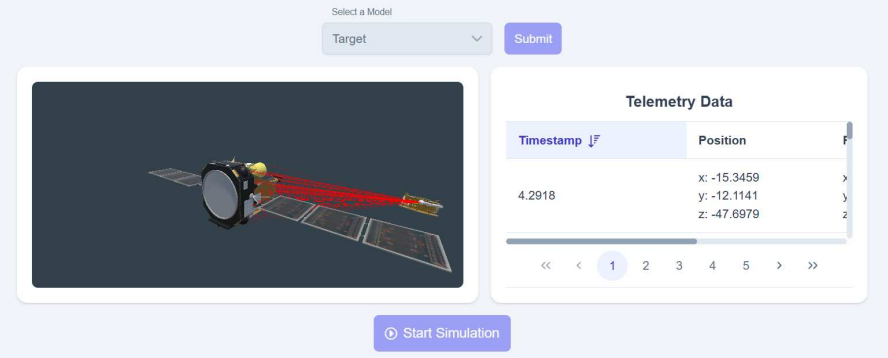


Figure 1.1: Example of a 3D simulation in a web application. On the left, a simulation of a satellite being scanned by a LiDAR sensor is visualized, while on the right, a table displays the telemetry data used by the simulation.

1.2 ALTEC

ALTEC (Aerospace Logistics Technology Engineering Company)

> "Is the Italian center of excellence for the provision of engineering and logistics services in support of operations related to the International Space Station (ISS) and the development of planetary exploration missions."

As described on the company website [1].

ALTEC operates with activities involving European Space Agency (ESA) and the German Aerospace Center (DLR), and also runs liaison offices at the National Aeronautics and Space Administration (NASA).

ALTEC plays a central role in several key areas of the aerospace sector.

- They provide engineering and logistics support for the operation and utilization of the International Space Station.
- They are involved in the design, development, and operation of mission control centers for robotic planetary exploration and re-entry missions, as well as data processing centers for space science missions.
- They support both scientific and commercial exploitation of various space platforms.
- They contribute to the advancement of future space utilization and exploration through a broad range of research activities.

ALTEC supports the ISS in various contexts. They provide support to the Italian Space Agency (ASI) for the operations of the Permanent Multipurpose

Module (PMM) on the ISS. In addition, ALTEC operates one of the Engineering Support Centers (ESC) for the ISS. They also operate as Ground Logistics Center, and they are also responsible for ISS Training Services to ESA.

One of ALTEC's major responsibilities in the context of planetary exploration is its participation in the ESA's *ExoMars* program. Specifically, they are in charge of designing, developing, and operating the Rover Operations Control Center (ROCC), which oversees both the system status and scientific operations of the ExoMars rover on the Martian surface.

In recent years, ALTEC has also become a significant contributor to space-based big data processing. They operate the Gaia Data Processing Center in Turin (DPCT), one of ESA's distributed data centers for the Gaia mission, which aims to map more than a billion stars in the Milky Way. They also participated in the Euclid mission, for which they manage the Italian Science Data Center.

ALTEC is also involved in the development and operational support of ESA's Space Rider program, an independent and reusable platform for access and return from low orbit. In this context, ALTEC acts as both the Payload and Landing Control Center, managing mission execution and vehicle recovery.

Beyond its operational roles, ALTEC participates in various research and development initiatives. These include areas such as virtual and augmented reality, machine learning, precision farming, small satellite operations and many more.

1.3 Goals

This thesis aims to develop a TypeScript library and two web-based software tools for supporting aerospace simulations, with particular emphasis on 3D visualizations and LiDAR 3D model reconstruction:

1. A TypeScript library that integrates Unity-based projects into a Vue.js web application, facilitating the use of interactive simulations directly in the browser.

2. A tool for reconstructing 3D models from LiDAR scans, a process that is crucial for aerospace operations. Additionally, a custom LiDAR scanning simulation has been developed to mimic a real-world scan.

3. A tool for visualizing anomalies in a scanned 3D model, comparing the scanned model with a known CAD model, helping to identify inconsistencies or structural issues.

1.4 Requirements

To ensure the integration of the tools into larger software ecosystems, which will be developed by ALTEC, a set of requirements served as the basis for their development.

First, all front-end sections of the tools must be easily integrable into the web application framework Vue.js. Vue is a JavaScript framework for building front-end applications. It was chosen because it is the front-end technology adopted by ALTEC, offers a performant, optimized rendering system that rarely requires manual tuning. Its versatility also makes it suitable for both modular integration and full-scale application development.

In addition, the tools and the library must be optimized to operate efficiently in web environments, minimizing computational overhead and memory usage to comply with the inherent limitations of WebGL applications. Particular care must be taken to ensure the simulations and visualizations remain lightweight enough to guarantee a fluid and responsive user experience.

Additionally, the tools are required to be modular and sufficiently general to allow their incorporation into larger and more complex systems. They are designed not as standalone applications, but as independent, reusable modules that can be adapted to a variety of operational contexts and project requirements. To support this modularity, the back-end components were containerized using Docker, enabling isolated execution environments and simplifying integration into different deployment pipelines. On the front-end side, particular attention was given to the generalization of both the Vue.js and Unity implementations, ensuring that the developed components could be easily reused and extended across multiple applications.

For the simulation and visualization components, the tools must maintain a low delay of interactions without excessive resource consumption. This involves efficient management of resources and careful optimization of shader programs, therefore ensuring compatibility with the constraints of web-based rendering pipelines.

The point cloud reconstruction tool, in particular, must process large-scale point clouds of 100,000 points. The entire reconstruction process should be completed within a reasonable amount of time when executed on a standard workstation.

Finally, the system integrating Unity-based simulations into the Vue.js web application must ensure reliable bidirectional communication between the Unity engine and the Vue.js front-end. It is required that this communication is capable of sustaining message exchanges at high frequencies, without introducing latency or causing any degradation in the simulation's frame rate or responsiveness.

1.5 Thesis structure

This thesis is structured as follows:

The chapter 2 presents the state of the art describing VR solutions in the web world, remeshing algorithms, testing frameworks for these algorithms, and methodologies for visualizing differences in 3D meshes.

The chapter 3 presents in detail the implementation, use, and testing of the library that incorporates Unity in Vue.

The chapter 4 presents the implementation of a framework to test remesh algorithms and the implementation of a LiDAR simulator.

The chapter 5 presents the implementation of the solution that visually shows structural defects of a scanned model compared to a reference CAD.

The chapter 6 describe the implementation of a reference demo that integrates the tools and the library.

The chapter 7 presents the conclusions and final analyses of the project and illustrates its possible future developments.

Chapter 2

State of the art

This chapter provides an overview of the current state of the art in the domains relevant to this work.

First, an overview of virtual reality technologies will be presented, with a focus on their integration in web environments.

Subsequent sections will examine the functionality of LiDAR systems and their use in aerospace operations and the implementations of virtual range scanning.

Analysis of surface remeshing algorithms will also be included.

Finally, a visualization of mesh differences with heatmap is analyzed.

2.1 Virtual reality and interactive applications

2.1.1 Virtual reality

Virtual Reality (VR) can be conceptualized as an interface to a plausible world where real and virtual elements are seamlessly integrated.

According to Burdea and Coiffet [2], VR is a high-level human-machine interface that leverages computer graphics for the real-time simulation of a realistic world. A crucial aspect of this simulation is the ability to interact with the elements within this synthetic environment through multiple sensory channels.

Burdea and Coiffet [2] define several characteristics inherent to this technology:

- **Interactivity**: The synthetic world presented in VR is dynamic and interactive. Interactions within this environment are instantaneous, meaning the virtual world reacts immediately to user stimuli. This immediate feedback is crucial as users perceive as real what they can influence and modify. The effectiveness of this perception is significantly enhanced when the metaphors for interaction closely mimic reality and are readily understandable by the user.

- **Immersion**: This characteristic quantifies the user's sensation of being physically present within the virtual world. To maximize the sense of immersion, it is essential to optimize computer graphics to render the environment as photorealistic as possible. Furthermore, involving other sensory modalities, including hearing, touch, and the visual system, substantially increases the user's perception of immersion. This concept is linked to the devices used by the VR application to interface with the user.

- **Presence**: While related to immersion, presence is a distinct concept that refers to the user's sensation of belonging to the virtual world. A greater sense of immersion generally correlates with a stronger sense of presence. However, immersion alone is insufficient to convince the user of the synthetic world's reality. For a robust sense of presence, the synthetic environment must react concretely and coherently to user stimuli. Additionally, the suspension of disbelief plays a fundamental role.

A distinction with immersive technologies lies in the classification of different forms of virtual reality, which vary primarily in terms of the level of immersion they provide to the user, as described in the Figure 2.1. This spectrum is effectively captured by the concept of the Reality–Virtuality Continuum, proposed in Milgram et al. [3], which describes a gradual transition between the real environment and a fully virtual one.

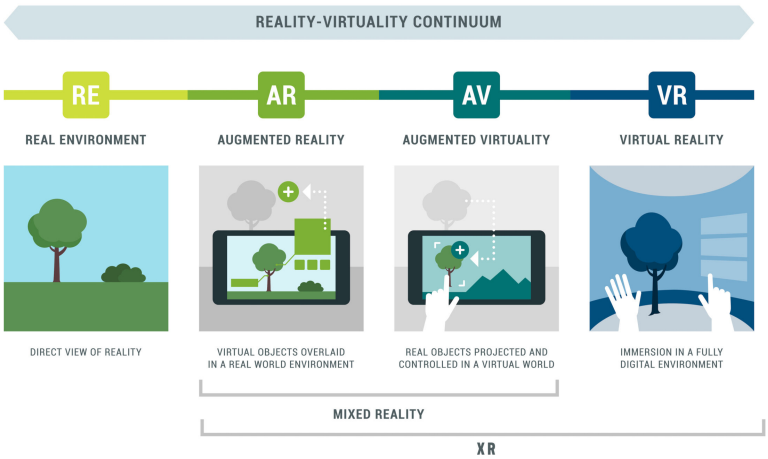


Figure 2.1: Virtual reality continuum. Image from https://creatxr.com/the-virtuality-spectrum-understanding-ar-mr-vr-and-xr/reality-virtuality-continuum-infographic-with-examples-2/

At the end closest to the physical world lies Augmented Reality (AR), where digital elements are superimposed onto the real environment, which remains the dominant frame of reference. Moving along the continuum, there is Augmented Virtuality (AV), where real-world components are embedded within a predominantly virtual environment, for instance, through the integration of physical objects or live video streams into a digital context.

Further along the continuum is Virtual Reality (VR) in the strict sense, in which the user is fully immersed in a computer-generated environment, with no direct perception of the surrounding physical world. Within this category, however, different subtypes exist based on the degree of immersion facilitated by the employed hardware. Notably, Desktop Virtual Reality refers to virtual environments experienced via conventional display screens (e.g., computer monitors) and standard input devices such as keyboards and mice. Although this form of VR offers a lower level of sensory immersion compared to head-mounted displays or spatial tracking systems, it remains the most widely adopted solution for interactive applications, particularly in fields such as education, design, simulation, and scientific visualization.

The focus of this thesis will be mostly on the field of desktop interactive applications.

2.1.2 Applications of immersive technologies in the aerospace context

The application of XR technologies represents a significant area of advancement in various aspects of space exploration, ranging from fundamental scientific discovery to the intricate processes of astronaut training, engineering design, and interactive operational support [4].

The following is an analysis of the applications of these technologies in space activities

Data visualization for scientific purposes

An example of VR applied to astrophysical research is NASA's *PointCloudsVR*[5], a custom 3D simulation used to visualize the motion and direction of approximately four million stars in the Milky Way. The Figure 2.2 shows an example of the *PointCloudVR* system used for a LiDAR scan.

This immersive visualization tool allowed researchers to visualize complex datasets, leading to enhanced classification of stellar groupings and the identification of previously misclassified stars. The spatial, interactive nature of the simulation made it possible to intuitively explore associations between stellar clusters, ultimately contributing to a more refined understanding of our local galactic

neighborhood. This application demonstrates the importance of simulations and visualization of computed data to help the analysis of such data.

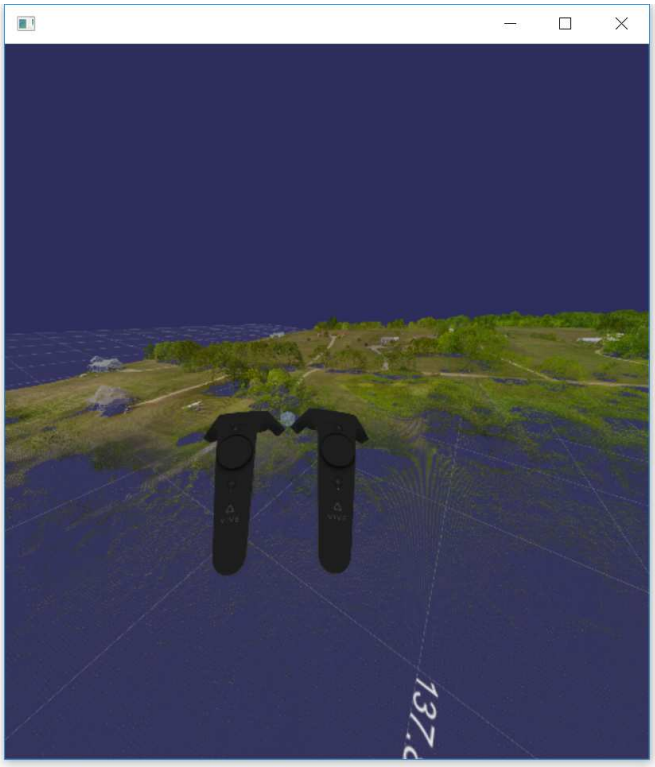


Figure 2.2: Example of *PointCloudsVR* showing a LiDAR point cloud VR. Image from https://github.com/nasa/PointCloudsVR

Astronaut Training

Immersive technologies are commonly used for astronaut training.

An example is the Virtual Reality Training Laboratory (VRL)[6]. It provides astronauts with detailed simulations of spacecraft and mission environments.

A central focus of the VRL is the training for extravehicular activity. These scenarios include practicing return maneuvers to the ISS after simulated detachment events, significantly improving astronauts' spatial cognition. The VR setup includes a physical hand controller synchronized with real-time simulations and head-mounted displays.

Another example of VR used for training is the Charlotte Mass Handling Robot[4], which simulates the manipulation of massive payloads in microgravity by incorporating force-torque sensors and realistic force feedback. This allows astronauts to gain practical experience in mass handling well before entering orbit.

ESA also employs VR for astronaut training through the JIVE system [7]. JIVE helps astronauts visualize the 3D configuration of robotic arms and understand how operations would occur in space. It also enables collaborative and remote teaching.

Real-time Operational Assistance

Immersive systems are increasingly employed to support operations and remote control in space missions. Such technologies are designed to provide information and alleviate the cognitive load on crew members by providing immediate, context-aware assistance.

Examples include *Sidekick*, *ARAMIS*, and the *Cold Atom Lab*, all of which leverage AR to overlay critical information directly into the astronaut's field of view [4]. This information may include holographic 3D schematics, procedural guidance for maintenance and repairs, or real-time inventory tracking.

Engineering Design, Prototyping, and Human Factors Analysis

In spacecraft and mission system design, VR enables early-stage visualization, iterative prototyping, and human factors evaluation without the need for costly physical mock-ups. Human Factors Engineering (HFE) greatly benefits from VR, as it allows engineers and users to experience and critique system layouts at an early stage[8].

Designing habitats such as Deep Space Habitats (DSH)[9] requires an understanding of how people will navigate and function in confined, microgravity environments. VR facilitates rapid scenario analysis, enabling engineers to test different configurations with full-scale models before committing to physical fabrication[8]. Similarly, VR is integrated into the development of the Space Launch System in Kennedy Space Center[10], particularly in evaluating ground operations and workspace ergonomics.

Digital Twins for Autonomous Systems and Damage Detection

Digital twins are increasingly central to the development of autonomous space systems. These virtual replicas of physical systems are continuously updated with telemetry data, sensor input, and simulation models to support diagnostics and decision making [11].

In the context of planetary rovers, digital twins serve as high-fidelity testbeds for developing Health and Usage Monitoring Systems (HUMS). These systems emulate real-world conditions providing an authentic testing ground for rover prototypes and the development of damage detection algorithms by injecting anomalies such as motor failures, thermal faults, or solar panel degradation[12].

NASA has developed digital and physical twins MAGGIE[13] and OPTIMISM[14] for its Mars rovers, which are used to test commands and study operational constraints.

Another example of the use of digital twins for planetary exploration is the DIGES project [12], which is modeled in MATLAB-Simulink with the aim to simulate lunar exploration scenarios. This model includes subsystems for structure, propulsion, power, and thermal control, offering a modular, adaptable platform for planetary exploration technologies.

2.1.3 Web-Based Simulation

Web-based simulation (WBS), defined as the integration of the World Wide Web with the field of simulation, has experienced significant growth and evolution[15].

As stated in Byrne et al. [15], Web-based simulation offers numerous advantages over traditional simulation systems, contributing to its growing appeal, such as ease of use, collaboration features, model reuse, ease access control and versioning, customization, and maintenance.

3D simulations in web pages are experiencing revolutionary changes with the rise of new web technologies. Its applications range from gaming to educational platforms and virtual events. Web-based simulation is expected to gradually replace mainstream 2D graphics technology, bringing revolutionary changes for future internet applications [16].

Past Integration Methodologies

In the early stages of Web-based simulation, various integration methodologies were explored. Java emerged as a dominant programming language for WBS due to its platform independence, reusability of classes, and web capabilities.

Important features provided by Java that made it suitable for WBS are:

- **JavaBeans Technology:** [17] JavaBeans facilitated the development of reusable software components that could be dynamically assembled using visual development tools. This allowed skilled programmers to create components (beans) that domain experts, with less technical expertise, could visually assemble into custom applications.
- **Platform Neutrality and Internet Capability:** Java simulations could be widely available, and applets could be retrieved and run without requiring porting or recompilation across different platforms [18]. This brought a high degree of dynamism to web applications.
- **Sophisticated Animations:** Java provided built-in support for animations, which was crucial for visualizing simulation results dynamically.

Several Java-based simulation environments and languages were developed, often as Java versions of existing simulation languages:

- **JSIM:** [19] A Java-based simulation and animation environment that allows models to be built as graphs. JSIM provides inherent animation capabilities and incorporates database connectivity for storing simulation results, based on the concept of Query Driven Simulation (QDS) [20]. It aims to provide a flexible, platform-independent, user-friendly environment for developing and running simulation models on the web.
- **SimJava:** [21] A process-based discrete event simulation package for building models of complex systems, including facilities for animated icons and integration into web documents.
- **JavaSim:** [22] A set of Java packages for building discrete event process-based simulations, implemented as a Java version of the C++ SIM toolkit [23].
- **Silk:** [24] A commercially available general-purpose simulation language implemented in Java, designed for building reusable modeling components and domain-specific simulators.

These early efforts laid the groundwork for future advancements in web-based graphics and computation. The general direction is shifting toward WebGL applications.

WebGL

Today, WebGL (Web Graphics Library) [25] stands out as a cross-platform, royalty-free open web standard for a low-level 3D graphics API, fundamentally based on OpenGL ES [26]. It allows compatible web browsers to render interactive 3D and 2D graphics without the need for plug-ins, directly within the HTML5 Canvas element. This technology supports GPU-accelerated rendering, which is crucial for handling intricate graphics and visualizations within web pages.

Key features of WebGL include:

- **Optimized Virtual Reality Experiences:** WebGL enables developers to create realistic and immersive virtual reality experiences directly in web browsers by leveraging hardware-accelerated 3D graphics.
- **Integration with Web Technologies:** It seamlessly integrates with HTML5, allowing interaction with DOM interfaces and other Web APIs. This enables developers to use existing web development skills for WebGL applications and combine them with features like data visualization, audio, and video.

- **Cross-Platform Compatibility:** WebGL operates across diverse devices and operating systems, including desktop computers, mobile devices, and virtual reality devices, facilitating unified VR experiences across platforms.

While WebGL offers significant capabilities, a newer, high-performance web graphics and computation API called WebGPU has emerged [27]. WebGPU provides web developers with JavaScript access to a user's device GPU for complex graphical and computational operations. WebGPU generally outperforms WebGL, exhibiting noticeably higher frame rates for similar applications, likely due to better hardware utilization and performance enhancements [28].

This indicates that WebGPU is a better choice for high-performance, graphics-intensive web applications.

In this thesis, WebGL was adopted due to its relative ease of integration. The simulations were developed using Unity, a platform that offers considerable flexibility in project export options, including seamless support for WebGL deployment.

Web Simulations in Aerospace Applications

Web-based visualization tools are proving increasingly valuable for aerospace simulations, providing efficient means to visualize complex system dynamics. These tools leverage advanced visualization capabilities to help engineers and developers understand simulation results and empower end-users with interactive experiences.

An example involves a generic web-based visualization tool specifically designed for aerospace simulations, utilizing the Cesium JavaScript library [29]. This tool benefits from the core advantages of web-based technologies, including platform independence, accessibility, maintainability, and interoperability.

Key features and capabilities of the above-mentioned web simulation tool include:

- **Integration with Simulation Tools:** They offer an interface for commonly used simulation tools and languages such as Matlab, Simulink, Scilab, and Octave, allowing commands to be sent from these platforms to the visualization application.

- **3D Visualization:** The tools provide rich, 3D representation of simulation outputs, which is critical for visualizing data from various domains including chemical, transportation, defense, and aerospace.

- **Real-time Data Integration:** These tools can receive real-time data from live systems via telemetry, radar, or similar technologies that transmit data using UDP or TCP messages. This data can also be recorded in a specific file format for later replay.

- **Distributed Architecture:** The tool is implemented with a distributed architecture composed of a visualization tool built in HTML and JavaScript, a web server, and a data server.

- **User Interaction and Customization:** Users can interact with the visualization to extend and refine models, observe the current state, change camera options, and modify settings.

- **Support for Multiple Platforms:** It can provide real-time visualization on various platforms, including desktop computers, mobile devices, and virtual reality glasses, offering immersive visual experiences.

2.2 LiDAR Scanning

Light Detection And Ranging (LiDAR) systems are widely employed for advancing autonomous capabilities in space missions, particularly for spacecraft relative navigation [30].

Their robustness and precision make them particularly suitable for aerospace applications.

In the context of this thesis, a virtual LiDAR simulator was developed to replicate the functionality of a physical scanner within a controlled simulation environment.

The following section presents an overview of the application of LiDAR sensors in aerospace scenarios, as well as existing approaches to virtual scanning for simulation and evaluation purposes.

2.2.1 Operating Principles of LiDAR

As explained by Christian and Cryan [30], a LiDAR is an active remote sensing device that fundamentally operates by illuminating a target with light, typically a laser, and measuring the time it takes for the emitted signal to return to the sensor, as shown in the Figure 2.3. This time-of-flight (ToF) measurement is then used to compute the range (r) from the sensor to a point on the target object using the formula:

$$r = \frac{c \cdot t}{2} \tag{2.1}$$

where c is the speed of light, and t is the laser time-of-flight.

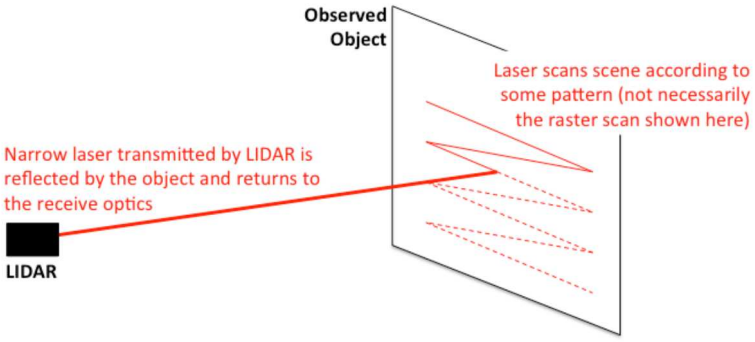


Figure 2.3: Representation of how LiDAR works

Classification of LiDAR Systems

LiDAR systems can be broadly categorized into three major groups based on their illumination and sensing mechanisms for a 3D scene:

Scanning LIDARs These sensors use a narrow laser beam that is swept over the sensor's FOV to obtain range measurements. The return from this laser typically illuminates a single detector. By combining knowledge of the laser's direction and the measured range, a three-dimensional (3D) point cloud of the scene can be constructed.

This type of lidar is simulated in the developed LiDAR virtual sensor.

Detector Array LIDARs These systems utilize a broad laser pulse to illuminate the entire scene at once (flood illumination). They then measure the laser ToF individually for each pixel on a detector array.

Similar to scanning LIDARs, a 3D point cloud is constructed by combining the known line-of-sight direction for each pixel with its associated laser ToF.

Spatial Light Modulator (SLM) LIDARs This emerging class of LIDAR, still under development, uses SLMs to sequentially illuminate subsets of the scene with a sequence of known patterns. The time history of the laser return from the entire scene (for each pattern) is then measured by a single detector.

By combining assumptions on scene geometry, an approximation of the 3D scene can be reconstructed from the time history of the returns.

2.2.2 Applications of LIDAR in Aerospace

LiDAR technology is extensively utilized in space applications, as described by Christian and Cryan [30], main application scenarios are:

- Autonomous servicing and refueling of existing orbital assets, e.g., the ISS.
- On-orbit assembly.
- Providing information on the relative state of two spacecraft, including distance and relative position and attitude.
- Autonomous rendezvous and docking.

Deployments of LiDAR in space missions

Historically, LIDARs have been successfully deployed on various spacecraft.

Some examples, provided by Christian and Cryan [30], are:

- **Trajectory Control Sensor (TCS):** the TCS is a scanning LiDAR equipped on the Space Shuttle. It was used as a relative navigation aid during rendezvous with the Mir Space Station, the Hubble Space Telescope (HST), and the ISS. Its use was instrumental in the development of the Rendezvous, Proximity Operations Program (RPOP).
- **Videometer (VDM):** Used by the ESA's Automated Transfer Vehicle (ATV), the VDM provides range, line-of-sight measurements for docking, and relative position and attitude. It operates similarly to a Flash LiDAR, using laser diodes to strobe the target and measure returns.
- **RendezVous Sensor (RVS)** The RVS is the primary relative navigation sensor on H-II Transfer Vehicle (HTV) resupply spacecraft, and it is also used for fault detection, isolation, and recovery on ESA's ATV. Both are scanning LiDARs that require retro-reflectors on the target.

2.2.3 Synthetic scanning

In the domain of surface reconstruction from point cloud data, virtual scanning techniques are employed to synthetically generate point clouds that accurately emulate the characteristics of real-world scan data. This approach is fundamental for benchmarking the performance of surface reconstruction algorithms.

Unlike uniform sampling, virtual scanning uses principles of range scans to capture sensor-specific characteristics.

An example of virtual range scanning implementation is provided by Berger et al. [31].

The virtual scanner simulates the functioning of an optical laser-based scanning system, incorporating both random and systematic error to realistically emulate the characteristics of real-world range scans. The methodology replicates physical laser scanning processes by projecting laser stripes onto an implicit surface and synthesizing the resulting radiance images.

To model random noise, the system first generates noise-free range data by ray tracing the implicit surface from a virtual camera's perspective, accounting for visibility constraints. Then it applies the random noise and the systematic noise models to generate the noisy data. A representation of the point cloud is provided in the Figure 2.4

The provided implementation, however, requires that the 3D model used in the scanning procedure meets a set of specifications:

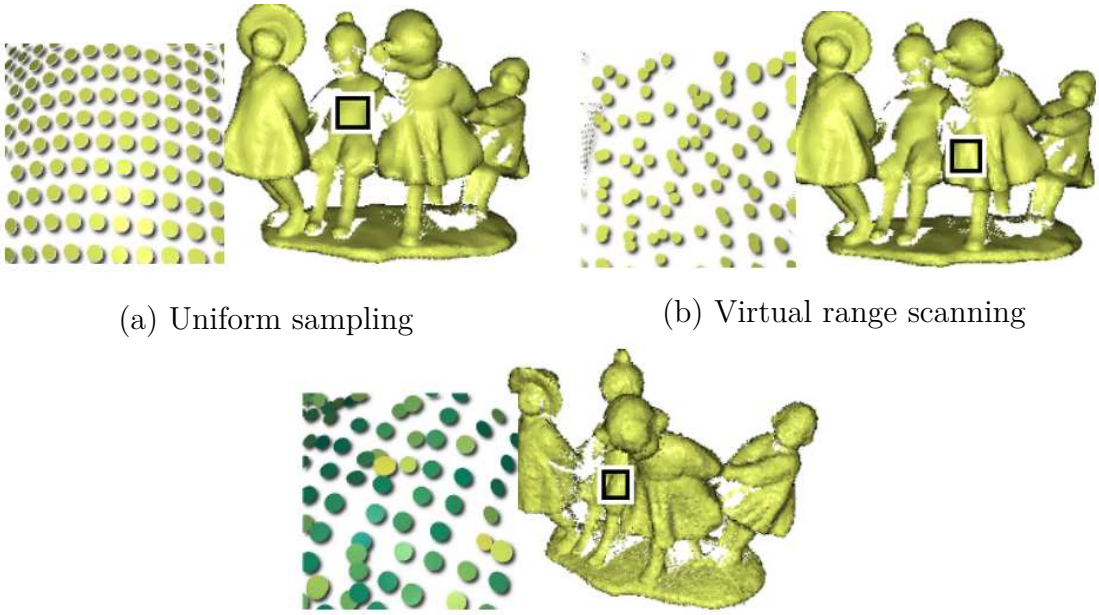
- **Watertight Surfaces**: To properly generate the scans, the mesh must be watertight. A watertight mesh is characterized by having no holes, thus uniquely defining an enclosed volume with a clear interior and exterior.
- **Manifold Surfaces**: Furthermore, the mesh must be a manifold surface. This means that the model is topologically well-behaved, implying that it does not contain problematic geometric ambiguities such as:
  - Self-intersections
  - Isolated vertices
  - Edges that are part of more than two faces

2.3 Remeshing algorithms

Surface reconstruction from an unstructured point cloud aims to generate a plausible surface that accurately approximates the input points. However multiple surfaces can potentially be generated from the same point data. Over the years, a wide range of approaches have been developed to address this problem. The following sections examine some of the algorithms in this field.

2.3.1 Screened Poisson Surface Reconstruction (SPSR)

SPSR [32] is an implicit surface reconstruction technique that builds upon the Poisson surface reconstruction algorithm. SPSR extends the traditional Poisson



(a) Uniform sampling

(b) Virtual range scanning

(c) Virtual range scanning with noise

Figure 2.4: Point cloud generated by different sampling techniques. Image from Berger et al. [31]

reconstruction by explicitly incorporating the input points as interpolation constraints. It reinterprets the surface reconstruction problem as solving a screened Poisson equation. The traditional Poisson method posits that the inward-pointing normal field of a solid's boundary can be viewed as the gradient of its indicator function, and it seeks a scalar function whose gradients best match a vector field derived from oriented point samples.

- SPSR augments the energy function of the original Poisson formulation with a screening term that penalizes deviations from zero at the sample points. This term makes surfaces smoother.
- The algorithm discretizes the system using B-spline basis functions, allowing for high-resolution detail near the surface.
- A multigrid approach is employed to efficiently solve the resulting linear system, progressing from coarser to finer octree depths.
- The implementation also supports Neumann boundary conditions, which are less restrictive than Dirichlet conditions, particularly when dealing with incomplete data.

2.3.2 Robust and Efficient Surface Reconstruction from Range Data (RESR)

The RESR[33] algorithm frames the surface reconstruction problem as an energy minimization task.

The initial step involves computing the 3D Delaunay triangulation of the merged range scan points.

The method minimizes an energy function by optimally labeling Delaunay tetrahedra as *inside* or *outside* the object. The energy function minimizes two terms:

- **Surface Visibility:** This term allows the correct labeling based on the approximate lines of sight.
- **Surface Quality Term:** Designed to prevent distorted triangles from forming part of the surface.

The reconstructed surface is defined as the boundary between tetrahedra with different labels, creating a watertight mesh.

2.3.3 Ball Pivoting Algorithm (BPA)

BPA[34] is a conceptually simple and efficient surface reconstruction technique that builds a triangle mesh by rolling a virtual ball over the point cloud.

The algorithm computes a triangle mesh that interpolates a given point cloud. Its fundamental principle is that three points form a triangle if a ball of a user-specified radius (ρ) can touch them without containing any other point. Starting from an initial *seed triangle*, the ball pivots around an edge, seeking a new point to form another triangle. The algorithm then incrementally grows the mesh: for each edge on the current mesh boundary, the ball pivots around it while maintaining contact with its endpoints. The pivoting motion is constrained, with the ball's center describing a circle. The first new point encountered by the ball during this motion forms a new triangle with the pivoting edge's endpoints.

To address issues like uneven sampling or holes, the BPA can be applied in multiple passes with increasing ball radii.

2.3.4 Advancing Front Surface Reconstruction (AFSR)

The AFSR[35] algorithm, implemented in CGAL, is a surface-based Delaunay method that iteratively selects triangles to grow a surface.

AFSR sequentially selects triangles to build a surface, using previously selected triangles to guide the selection of a new triangle. This greedy selection aims to

generate an orientable manifold triangulated surface that plausibly approximates the input points.

The process begins with the construction of a 3D Delaunay triangulation of the input point set. The algorithm initializes its surface with the triangle that has the smallest radius. The boundary of this initial triangle becomes the *advancing front*. At each step, the algorithm extracts the most plausible candidate triangle adjacent to the current surface's boundary. New candidates are dynamically selected as new boundary edges appear

Strict topological constraints are enforced to ensure the output is a manifold surface.

The algorithm includes heuristics to discard uneven triangles.

2.3.5 Scale-Space Surface Reconstruction (SSSR)

SSSR[36] is a CGAL package that takes an unordered point set as input and computes a triangulated surface mesh that interpolates the points. This method is able to manage noise and outliers.

SSSR creates a reconstruction using a scale-space. Increasing the scale smooths the underlying surface described by the point set. The overall process involves reconstructing the point set at a coarse scale and then reverting the mesh's points back to their original locations. The method operates in two main phases:

- **Scale-Space:** The point set is smoothed by iteratively increasing the scale using *Weighted PCA smoother* and *Jet smoother* operator
- **Meshing:** After smoothing, a triangulated surface mesh is computed with *advancing front mesher* at the coarse scale, and the connectivity is propagated back to the original point cloud.

2.4 Mesh Difference Visualization

In the scope of this thesis, accurately quantifying geometric discrepancies between 3D meshes is a fundamental task to correctly implement the mesh difference visualization tool. Several techniques have been proposed to compute the distance metrics between geometries. As discussed in Ren et al. [37], among the most common approaches are *Point-to-Point* (P2P) and *Point-to-Face* (P2F) distance metrics.

2.4.1 Point-to-Point Distance

Point-to-Point (P2P) metrics typically operate by sampling both meshes into point clouds and computing distances between the corresponding points. Two widely

adopted metrics in this category are the Earth Mover's Distance (EMD)[38] and the Chamfer Distance (CD)[39]. These methods establish pairwise correspondences between points and compute aggregate distances across the entire surface. As described in Ren et al. [37], while effective in many contexts, P2P methods can suffer from significant limitations due to their reliance on discrete sampling. This approach disregards the continuous nature of the surface, which may result in incomplete or misleading estimations.

2.4.2 Point-to-Face Distance

Point-to-Face (P2F)[40] metrics present an alternative strategy. In this approach, points are sampled from one surface, and the shortest distance from each sampled point to the faces of the opposing mesh is computed. The aggregate of these minimum distances provides a measure of surface dissimilarity. P2F methods reduce some limitations of P2P techniques by avoiding point-to-point correspondence and making better use of the mesh geometry.

2.4.3 Heatmap-Based Visualization Tools

A practical implementation of mesh comparison is presented in Rodriguez-Garcia et al. [41], which introduces a Blender plugin for visualizing geometric discrepancies using color-coded heatmaps, as shown in Figure 2.5. The plugin supports the analysis of distance deviations between two meshes. The computed distances are encoded as vertex weights and visualized in weight painting mode or exported as texture attributes. The heatmaps use customizable color palettes, and thresholds are adjustable according to the model scale.

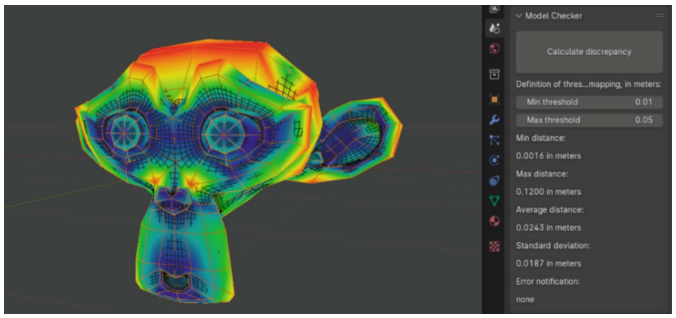


Figure 2.5: Heatmap representation of the work proposed in Rodriguez-Garcia et al. [41]. Image from [41]

The plugin operates by comparing each vertex of the low-resolution mesh against the high-resolution counterpart. For each vertex in the low-poly mesh, a cone is projected along the normal in both directions to identify a corresponding point on

the high-poly mesh. If no intersection is found, the cone aperture is increased up to a threshold, after which a default distance of zero is assigned. The resulting distances are mapped to a [0, 1] range using user-defined thresholds, and the data is stored in a new vertex group.

In this thesis, a heatmap-based method for mesh comparison was developed, inspired by the concepts presented in the aforementioned plugin and literature. However, instead of relying on vertex-based comparison, the technique implemented in the thesis work uses a uniform resampling of the mesh surfaces to compute distances. This approach improves the robustness and accuracy of the comparison, as it avoids dependency on the mesh resolution or vertex layout.

Chapter 3

Unity in Vue.js

One of the earliest steps in developing 3D simulations or visualizations in a web-based environment is the integration of a game engine such as Unity into the application front-end. In the context of this thesis, the front-end architecture is Vue.js, as ALTEC uses this framework for its projects. A custom TypeScript library was developed to allow the embedding of a Unity build into a Vue.js web application. The primary goals of the library are to manage the lifecycle of the Unity instance and enable bidirectional communication between the Vue front-end and the Unity engine.

3.1 Motivation and Design Considerations

The architectural separation of concerns led to the choice to keep the responsibilities for data management, user interaction, and API connection in Vue.js while isolating the simulation logic within Unity. The front-end system is responsible for all aspects of UI rendering, API communication, authentication, and state management. Integrating part of this logic within Unity would lead to redundancy, potential security issues (e.g., duplicate authentication handling), and a less maintainable codebase.

Furthermore, communication between the two systems is important, for example, in aerospace simulation scenarios, such as those involving the visualization of telemetry data. In this scenario, the Vue application handles API calls to retrieve telemetry data from a back-end service. Once received, these data must be passed to Unity to update the simulation in real time. In the opposite direction, Unity may also need to emit events such as simulation states, user actions, or error messages that the Vue interface must handle appropriately.

For these reasons, the integration requires a lightweight communication bridge that provides interactive message exchange between the two environments while

minimizing performance overhead.

3.2 Requirements

The development of the library was guided by a set of requirements.

- **Lightweight footprint:** The library must avoid the inclusion of unnecessary external dependencies and keep its codebase minimal to reduce the size of the final web application bundle.
- **Minimal overhead:** Communication between Vue and Unity should not introduce noticeable latency or consume excessive system resources.
- **Bidirectional communication:** The library must provide APIs to:
  - Send messages from Vue to Unity (e.g., telemetry data updates or simulation instructions).
  - Receiving events from Unity in Vue (e.g., simulation events or interaction callbacks).
- **Unity Lifecycle Management:** The library must handle the lifecycle of a Unity instance embedded within a Vue component. This includes initialization, loading, execution, shutdown, and cleanup of the Unity environment.

3.3 Library Functionality

The core functionalities of the library are summarized as follows.

- **Unity Loader Component:** A Vue component that dynamically loads a Unity WebGL build from a specified URL and embeds it into the DOM. This component handles the initialization of the Unity instance.
- **Message Channel:** A messaging interface that allows Vue components to send messages to Unity and to listen to events from it.
- **Lifecycle Manager:** APIs that control the Unity instance's lifecycle. Including methods to initialize, retrieve loading status, dispose of the Unity runtime, and ensure proper resource cleanup.

3.4 Implementation Details

This section describes the implementation of the TypeScript library.

Unity supports the generation of WebGL builds that can be executed directly in a web browser. A typical Unity WebGL build consists of several files: a `.data` file containing binary assets, a `.framework.js` file responsible for running the Unity engine logic in JavaScript, a `.wasm` file that holds the compiled C# code in WebAssembly format, and a `.loader.js` script that bootstraps the Unity instance in the web page. Taking advantage of this functionality, the library will use the compiled WebGL application and instantiate a `canvas` element with the Unity instance.

To keep the bundle size of the Vue.js integration library minimal, advanced abstractions such as Vue's Composition API were avoided. Instead, a low-level implementation using the `defineComponent` function and the `h()` rendering helper was adopted. Given that the component's purpose is to render a single HTML canvas element for the Unity simulation, this choice appears simple enough to be implemented with low-level functions.

The following subsections describe in detail how Unity is loaded into the Vue application, how the communication channel is structured, how resource are cleaned up, and how mouse and keyboard focus handling are implemented.

3.4.1 Usage Overview

Once integrated into a Vue.js project, the library is used via a dedicated component named `<UnityComponent />`. This component acts as the interface between the Vue application and the Unity WebGL build. Configuration parameters related to the Unity build must be provided to the component. Additionally, the component supports lifecycle event hooks to monitor the initialization process of the Unity instance.

An example of how the component is used within a Vue template is shown in the Listing 3.1

The emitted lifecycle events are:

- **beforeLoad** — Emitted immediately before the Unity WebGL instance begins to load. This can be used to display loading indicators or temporarily disable UI elements.
- **progress** — Continuously emitted during the loading phase, providing a numeric value between 0 and 1 that indicates the current progress. This allows the application to render progress bars or other visual feedback.
- **loaded** — Emitted once the Unity instance has successfully completed its loading process and is ready for interaction. This event provides access to

a `UnityChannel` object, which is a communication interface with the Unity engine.

- **error** — Emitted if an error occurs at any point during the loading phase. The associated payload includes an `Error` object with the error's details.

The `UnityChannel` object, provided through the `loaded` event, exposes methods to send messages, subscribe to Unity-generated events, and terminate the simulation.

```
1 <UnityComponent
2   :unityConfigs="configs" width="950px" height="600px"
3   @beforeLoad="onBeforeLoad"
4   @progress="onProgress"
5   @loaded="onLoaded"
6   @error="onError"
7 />
```

Listing 3.1 Example of UnityComponent usage

3.4.2 Unity Loading

The Unity engine provides official support for WebGL builds, which can be embedded and executed directly within a browser, as described in the Unity documentation [42]. To load a Unity WebGL project on the client side, Unity generates a set of JavaScript and binary files during the build process. Among these, the `loader.js` file exposes the `CreateUnityInstance` function globally, which initializes the Unity runtime and attaches the instance to a given HTML canvas element.

In the context of this library, it cannot be assumed that the host application has already included the `loader.js` script in the HTML document. For this reason, the script must be loaded dynamically at run-time. To achieve this, the library creates a new `<script>` element using the standard DOM API as shown in the listing 3.2

Once the script is successfully loaded, the globally available `CreateUnityInstance` function becomes accessible. This function is then invoked with the target canvas element, the configuration object, and a callback function used to report the progress of the loading operation.

The loading process is summarized as follows:

- The canvas element is created with the `h()` function, and a reference to it is stored in the Vue state.
- The `CreateUnityInstance` function is called with the canvas and the Unity build configuration.
- Unity begins loading the WebAssembly and data files. During this phase, the provided callback is invoked repeatedly with a progress value between 0 and 1.
- Once the loading is complete, the function resolves and returns a reference to the Unity instance.
- If any error occurs during this process (e.g., missing files or incompatible browser), an exception is thrown and must be caught to handle the failure gracefully.

This runtime loading approach ensures that the Unity environment is only initialized when needed and does not require the host application to modify its HTML structure or statically include Unity scripts.

```
1 const script = document.createElement('script');
2 script.src = unityLoaderUrl;
3 [...]
4 document.body.appendChild(script);
```

Listing 3.2 Creation of script tag in JavaScript

However, the following issues can emerge:

- **Duplicate script insertions:** Each time the Unity component is mounted, a new `<script>` tag pointing to the Unity loader may be inserted into the document. If not properly managed, this can lead to the accumulation of redundant scripts.
- **Global scope conflicts:** The `CreateUnityInstance` function is injected into the global scope when the loader script is executed. If multiple Unity WebGL builds are used within the same application, especially builds generated from different Unity versions, then the global function may be overwritten or point to the wrong version, causing compatibility issues.

To mitigate these problems, the library implements a custom strategy to integrate the loading scripts.

- When a loader script is requested for the first time, the library dynamically creates and appends a `<script>` tag as described earlier. Afterwards, the corresponding `CreateUnityInstance` function is stored in a map, which associates loader URLs with their respective function references, and deleted from the global scope.
- Before attempting to load any new script, the library checks whether the requested loader URL has already been loaded. If it has, no new script tag is added to the document, and the previously cached `CreateUnityInstance` reference is reused.
- When initializing a Unity instance, the library retrieves the correct function for the specific loader that was used, ensuring that the instantiation process is aligned with the appropriate Unity build. This prevents accidental use of an incorrect global function.

This mechanism guarantees that each Unity WebGL build is correctly loaded.

3.4.3 Communication Channel

Once the Unity instance has been successfully initialized, the library provides a communication interface called `UnityChannel`. This object serves as an abstraction layer for bidirectional messaging between the Vue application and the Unity simulation. Its purpose is to simplify interaction by encapsulating the Unity WebGL messaging API and exposing a set of methods to send commands and receive events.

The `UnityChannel` interface implements the following methods:

- **Method Invocation:** The `call()` method allows Vue to invoke methods defined in Unity scripts. It targets a specific Unity GameObject identified by name and calls the specified method, optionally passing parameters. This mechanism is built on top of Unity's standard `SendMessage` API.
- **Event Subscription:** The `subscribe()` method allows the Vue application to listen for events triggered from Unity. When Unity emits a message (via a JavaScript function call), all registered subscribers for the corresponding event are notified. The `unsubscribe()` method can be used to remove a registered event listener.
- **Instance Termination:** The `quit()` method shuts down the Unity instance, releases associated resources, and clears all active event subscriptions.
- **Fullscreen Control:** The `setFullscreen()` method allows the Vue application to toggle Unity's fullscreen mode.

Implementation Details of Unity Channel

In order to support communication between the Unity WebGL instance and the Vue.js application, it is necessary to establish a shared communication layer accessible from both environments. Since Unity WebGL builds communicate with the browser via JavaScript through custom `.jslib` file, a global object is exposed to serve as the bridge between the two runtimes.

The library defines a global communication object and attaches it to the `window` object using a reserved namespace. This object acts as a centralized message bus, allowing Unity to emit events that can later be intercepted by the Vue application. As long as both Unity and Vue are aware of the global namespace, they can exchange messages.

Internally, the communication layer is composed of two classes:

- **`UnityGlobalChannel:`** This is the actual global object exposed under the reserved namespace in `window`. It is designed to be accessed only by Unity, allowing Unity-side scripts to publish events to the global channel. It maintains a mapping of subscribers and broadcasts messages to all registered listeners for a given event name

- **`UnityChannel:`** This is the object provided by the Vue component once Unity is initialized. It serves as the main entry point for the Vue application to communicate with Unity. Internally, it encapsulates a reference to the corresponding `UnityGlobalChannel` instance and exposes methods for event subscription, unsubscription, and method invocation.

This approach allows Unity to emit events through the `UnityGlobalChannel`, while Vue interacts with the `UnityChannel` interface. This design prevents misuse of internal methods.

Unity Channel Usage

The following is an explanation of how to use the `UnityGlobalChannel` and `UnityChannel`.

Calling Unity from Vue The `UnityChannel` object is provided through the `loaded` lifecycle event of the Vue component. Through this object, the front-end can invoke methods defined within Unity simply by using the `SendMessage` function.

The workflow is described in the top section of the Figure 3.1.

Receiving Events from Unity in Vue The communication channel is also designed to receive events from Unity. In WebGL builds, Unity allows communication with the browser's JavaScript context via `.jslib` files. These JavaScript files act as bridges, allowing C# scripts to trigger functions defined in JavaScript.

To send data back to Vue, Unity calls a function in the `.jslib` file, which then interacts with the global JavaScript object managed by the library. This object is automatically created by the library and corresponds to an instance of the internal `UnityGlobalChannel` class. This global instance exposes a `publish` method, used to emit events to the Vue application.

On the Vue side, the `UnityChannel` exposes a `subscribe` method that allows the registration of one or more callback functions to a specific event. When Unity publishes an event, all registered callbacks are executed, and any return value from the last callback can be passed back to Unity if needed.

The workflow is described in the bottom section of the Figure 3.1.

Unity Script to Simplify Communication

A custom Unity script is developed to reduce the complexity of event publishing. In standard WebGL builds, Unity requires a separate JavaScript function, defined in a `.jslib` file, for each event that needs to be sent to the JavaScript context. These functions must also handle data conversion (e.g., converting C# strings or arrays to valid JavaScript formats), leading to the creation of several similar functions for every event.

In fact, a C# helper class named `JsChannelBinding` was implemented. This class acts as an abstraction layer over the raw implementation, allowing Unity to emit events with reduced complexity. The class `JsChannelBinding` exposes a set of methods that allow the publishing of events to JavaScript. These methods accept a single parameter, which may be of type `int`, `float`, `string`, `float[]` or any serializable C# object. In the case of complex objects, the class handles JSON serialization on the Unity side; the corresponding listener on the JavaScript side receives the data already deserialized. The complete communication workflow is described in Figure 3.1

Dynamic Channel Binding Since the global JavaScript object name may vary depending on the configuration passed to the library from Vue, the class must be initialized with the correct channel name. The recommended way to achieve this is to first transmit the channel name from Vue using the `call` method exposed by the `UnityChannel`. Unity then stores this value and uses it when instantiating the `JsChannelBinding` object, ensuring that all published events are correctly routed to the corresponding Vue instance.

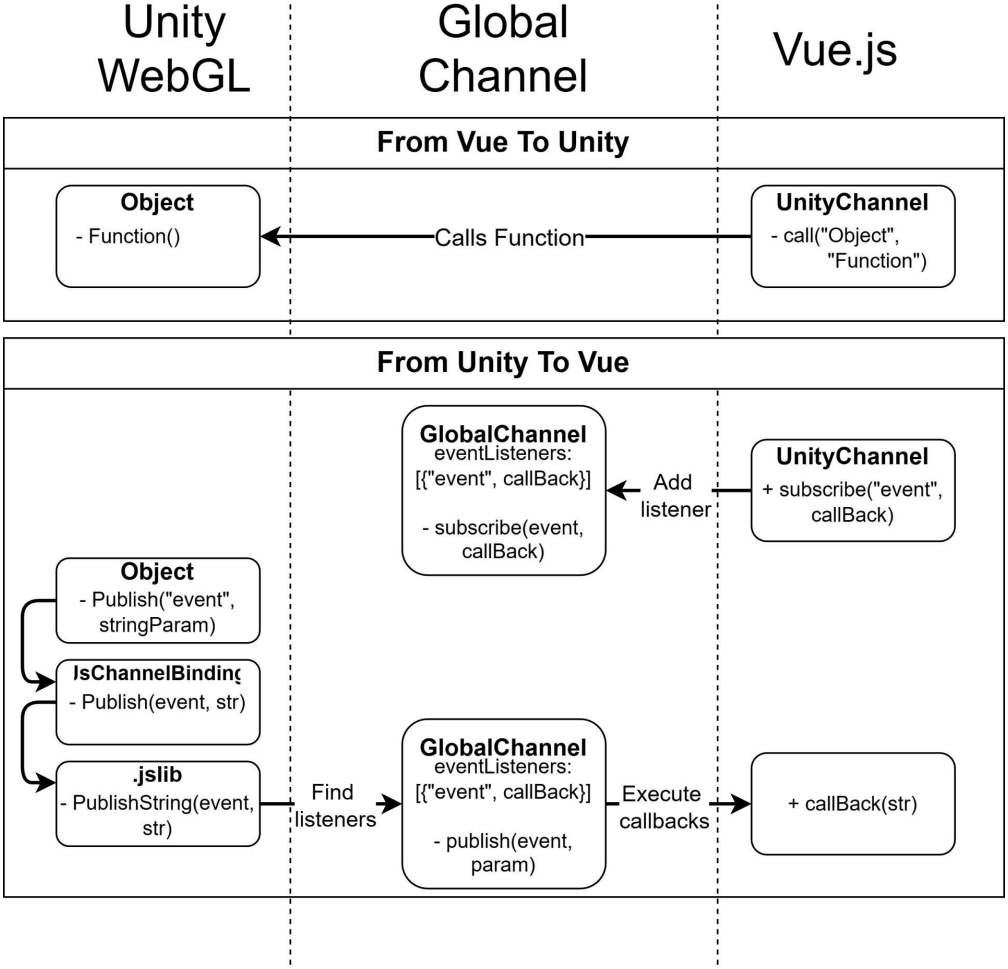


Figure 3.1: High level communication workflow between Unity and Vue

This configuration is important in projects where simulations may run concurrently on the same page. The ability to bind to a specific channel avoids cross-communication. Other Solution to the multiple instance problem are described in the section 3.5

Cleanup

To ensure proper resource management and avoid memory leaks, a dedicated cleanup function has been implemented in the `UnityChannel` class.

The `UnityChannel` class exposes a `quit()` method, which is responsible for terminating the Unity simulation and releasing all associated resources. The cleanup process manages the destruction of both the Unity WebGL canvas instance and

the underlying communication channel. Moreover, the `UnityChannel` object itself must no longer be usable after the cleanup.

The first step in the cleanup process involves invoking the `Quit` method provided by the standard Unity API in the `UnityInstance` object.

The `Quit` method initiates the shutdown of the Unity engine and deallocates its internal resources, including the rendering context attached to the canvas element.

Subsequently, the global communication channel is cleared. This involves unregistering all event listeners bound to the channel. Since the global channel is implemented as a global variable, clearing it is a crucial step not only to free memory, but also to avoid unintended behaviors when initializing a new Unity instance.

Despite the cleanup of Unity and its communication channel, the `UnityChannel` object remains available to external components, as it cannot be explicitly destroyed from within the library's scope (e.g., via `delete`). The private flag `isActive` is used to prevent further use of the invalid object. This flag is set to `false` when the `quit()` method is invoked. All public methods of the class check the state of this flag before executing any logic, and if `isActive` is `false`, an error is thrown. This mechanism ensures that any subsequent attempt to use the `UnityChannel` after cleanup results in a failure, providing a signal to the developer that the object is no longer in a valid state.

Finally, the canvas element must be destroyed. This is handled with the use of the render functionality of Vue. Vue components are rendered programmatically at every state change, this is the default reactive behavior of the framework. Leveraging this functionality, it is possible to render the `canvas` element conditionally: when the channel is active, the `<canvas>` element is rendered, otherwise an empty element is displayed. This mechanism ensures that the unused HTML element is removed from the page, cleaning up memory and avoiding showing a freeze-frame of the simulation.

Mouse and Keyboard Focus

In simulations, user inputs via mouse and keyboard are essential for interaction within the simulated environment. However, in the context of the developed library the simulation typically coexists with other user interface components of the web page (e.g., text inputs, buttons, form elements), which also rely on the same input devices.

By default, Unity WebGL builds capture all keyboard input and lock the cursor to the canvas element when it receives focus. This behavior can interfere with the functionality of other UI components by redirecting all keystrokes to the Unity instance, regardless of the user's actual intent. To avoid such interference, a specific focus management is needed.

The core idea is to restrict mouse and keyboard input to the Unity simulation only when the user explicitly interacts with it, and to release control when the user either presses the `Esc` key or interacts with another element outside the simulation canvas.

On the implementation side, the Vue library assigns a default `tabindex` value of `-1` to the Unity canvas element. This makes the canvas focusable while preventing it from being part of the normal tab navigation order. The `tabindex` can be overridden via a component prop if needed.

To enable proper keyboard delegation, the Unity WebGL application must disable global keyboard capture by setting:

```
WebGLInput.captureAllKeyboardInput = false;
```

This directive should be issued as early as possible in the Unity application lifecycle, ideally within the `Start()` method of the main controller script. Additionally, to prevent the cursor from being locked upon interaction with the canvas, the sticky cursor behavior can be disabled by setting:

```
WebGLInput.stickyCursorLock = false;
```

With this setup, the Unity instance captures input only when active, preserving the expected behavior of other UI elements on the web page.

3.5 Handling multiple Unity instances

An important aspect of integrating Unity WebGL into a Vue web application using the developed library is the ability to manage multiple Unity instances running at the same time on the same web page. The resolution of this issue requires adjusting the default usage of `UnityChannel`.

Due to the fact that the `GlobalUnityChannel` is injected into the `window` as a global object, all Unity instances running on the same page will access and use the same communication object. As a result, it is not possible to distinguish the origin of the Unity instance that emits an event when multiple Unity builds use the same event names.

3.5.1 Isolated Channels for Different Unity Projects

One strategy to mitigate these conflicts, particularly when dealing with different Unity projects (i.e., builds generated from distinct Unity projects), is to assign each instance a unique channel name. This is achieved by modifying both the Vue.js configuration and the Unity WebGL `.jslib` file.

On the front-end side, each Unity component is provided with a dedicated configuration object that specifies a unique `channelName`. An example is reported in Listing 3.3

```
const unityConfigs1: UnityConfigs = {
  channelName: '_FIRST_PROJECT_'
  ...
}

const unityConfigs2: UnityConfigs = {
  channelName: '_SECOND_PROJECT_'
  ...
}
```

Listing 3.3 Assigning unique channel names in Vue.js

In the Unity `.jslib` files, events must be routed through the specific channel, as shown in the Listing 3.4.

```
mergeInto(LibraryManager.library, {
  example: function (str) {
    window._FIRST_PROJECT_.publish('example', str); // for
   the first instance
  },
});
```

Listing 3.4 Using distinct channels in Unity .jslib

This setup ensures that each Unity instance communicates exclusively through its own isolated channel, avoiding naming collisions and ensuring correct event routing.

3.5.2 Managing Multiple Instances of the Same Project

A more complex scenario appears when multiple instances of the same Unity project (generated from the same source code and build configuration) must run

simultaneously on the same page. In this case, the assignment of distinct channel names is not feasible, as the underlying WebGL build is identical.

To address this, upon initialization, Vue.js sends a unique postfix string to each Unity instance, which is then used to differentiate events at runtime. This approach involves calling a specific method in Unity to send the postfix and adjusting the subscription to account for the modified event name; an example is provided in Listing 3.5

```
1 channel.call("JSCommunicationHandler", "SetEventPostfix",
    postfix);
2 channel.subscribe('HelloJS' + postfix, () => { alert('Game
    Start') });
```

Listing 3.5 Subscribing to events with a unique postfix

On the Unity side, a singleton component named `JsCommunicationHandler` stores the received postfix and exposes it to other parts of the application.

In the `.jslib` code, every published event must include this postfix, as shown in Listing 3.6

```
1 mergeInto(LibraryManager.library, {
2   HelloJS: function (postfix) {
3     postfix = UTF8ToString(postfix);
4     window._UNITY_CHANNEL.publish('HelloJS' + postfix);
5   },
6 });
```

Listing 3.6 Publishing events with a postfix in .jslib

Finally, Unity scripts invoking these JavaScript functions use the stored postfix as a parameter.

This solution allows multiple Unity instances to coexist and communicate independently without interference.

3.6 Performance testing

This section details the methodology and results of the performance testing conducted on the developed library. The primary goals of this evaluation are to assess performance indicators such as memory usage, Unity's frame rate and the number of messages that could be sent per second between the Vue application and the Unity instance with regard to the default implementation of WebGL builds generated by Unity.

For web-specific performance, Google Lighthouse is a widely recognized de facto standard for testing web pages performances, providing performance metrics, such as page loading information. However, a more specific analysis of individual features provided by the developed library was required for this specific evaluation. Performance metrics were also collected using the Microsoft Edge performance monitor tool. This tool, in addition to the web vitals metrics, also provides a representation of memory and CPU usage, as described in the official documentation [43] .

3.6.1 Test Objectives

The performance evaluation aimed to achieve the following specific objectives:

1. Compare memory usage and frame rate across different configurations.
    - A standalone HTML file generated by the Build and Run function of Unity editor (serving as a control).
    - A minimal Vue project integrating Unity using the developed library.
    - A more realistic Vue project that incorporates a Unity project along with additional UI elements, such as forms and buttons.
2. Evaluate message throughput:
    - Measure the maximum number of messages that could be sent from JavaScript (Vue) to Unity per second without negatively impacting the game's performance.
    - Measure memory usage and frame rate while utilizing different implementations for the global communication channel responsible for handling event listeners from Unity to Vue.
3. Evaluate different implementations of event listener handling.

3.6.2 Unity Project Setup and Hardware

The Unity scene used for testing was designed to represent a scenario that involved complex environments and physics. The scene included a third-person controllable character, an interior environment with multiple light sources and assets (such as chairs and tables), and interactable physics cubes, as shown in Figure 3.2.

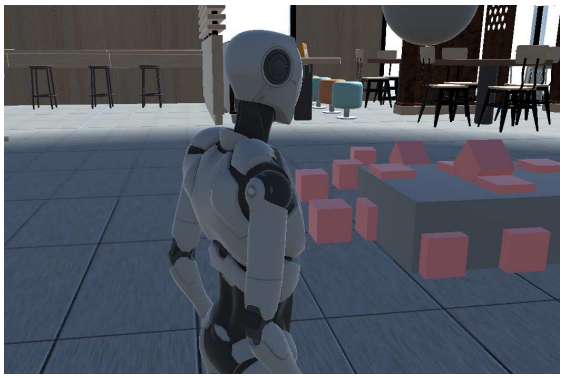


Figure 3.2: Unity simulation used to conduct the tests.

The tests were conducted using Microsoft Edge as the browser environment. The hardware characteristics of the test machine included an Intel i7-8750H processor and an NVIDIA GeForce RTX 2060 graphics card.

3.6.3 General Performance Testing

Different scenarios were created to evaluate performance variations compared to the default Unity build, which served as control.

Unity Build (Control)

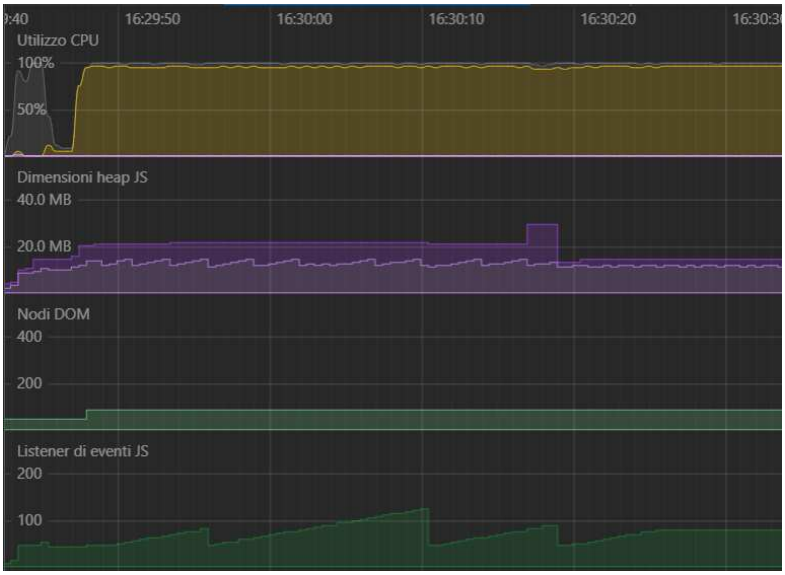
A production build of the Unity project was run directly using the "Build and Run" option in Unity, configured to open in Microsoft Edge.

Key metrics observed for this control scenario included:

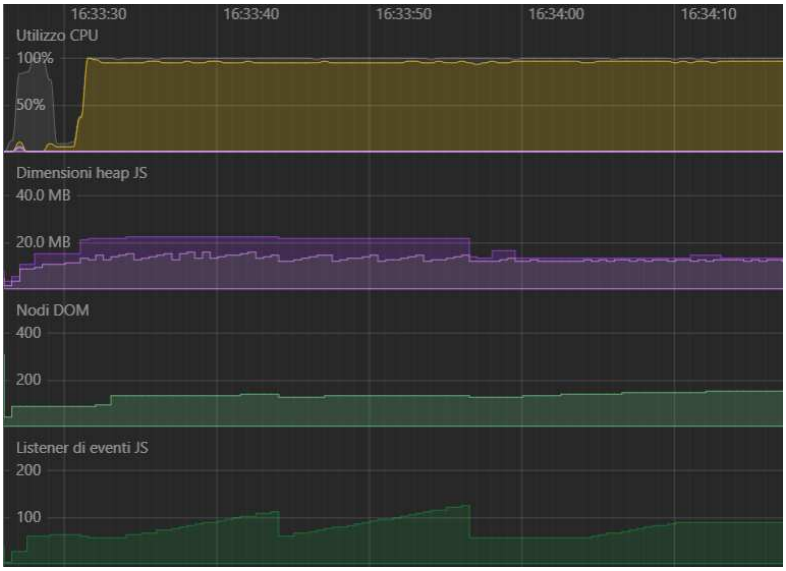
- Time to load the page and start the game: 5.04 seconds.
- Peak heap size: 10.9 MB.
- Total network loading time: 1010 ms (The Unity Wasm and data files required the longest loading time, highlighting that the majority of the time is dedicated to Unity build files).
- CPU usage, as shown in the Figure 3.2: 100%.

- Maximum memory used, as shown in the Figure 3.2: 20 MB.
- Framerate: Stable at over 100 fps.
- GPU memory usage: Maxed at 537 MB.

(a) Unity Build



(b) Minimal Vue project



(c) Real-like Vue Project

Figure 3.2: Performances recorded using the Microsoft Edge's performance tool: (a) performances of the page generated by the Build and Run process of the Unity editor; (b) performances of a minimal Vue project that uses the library in analysis; (c) performances of a real-like Vue project that uses the library in analysis. This image highlights that there are few to no differences among the recorded data.

Minimal Vue Project

This scenario involved a Vue project containing only an App component configured to initialize the Unity build using the developed library.

The metrics for the minimal Vue project included:

- Time to load the page and start the game: 5.1 seconds.
- Peak heap size: 13.8 MB.
- Total network loading time: 1050 ms.
- CPU usage, as shown in the Figure 3.2: 100%.
- Maximum memory used, as shown in the Figure 3.2: 20 MB.
- Framerate: Stable at over 100 fps.
- GPU memory usage: Maxed at 540 MB.

Instantiating Unity within a Vue project using the developed library was observed to not introduce significant overhead. All the recorded metrics are almost identical to those of the control scenario, with a slightly higher heap size and networking time, likely due to Vue overhead.

Real-like Vue Project

This test scenario expanded upon the minimal Vue project by including additional UI elements within the Vue template, such as buttons and forms.

The metrics for the real-like Vue project included:

- Time to load the page and start the game: 5.5 seconds.
- Peak heap size: 12 MB.
- Total network loading time: 1080 ms.
- CPU usage, as shown in the Figure 3.2: 100%.
- Maximum memory used, as shown in the Figure 3.2: 20 MB.
- Framerate: Stable at over 100 fps.
- GPU memory usage: Maxed at 572 MB.

The addition of UI elements was found to have no significant impact on performance. In fact, the recorded values are almost identical to the minimal Vue scenario.

3.6.4 Multiple Unity Instances

To assess the performance as the number of Unity instances on the page increases, the tests described in this subsection were carried out.

The tests were conducted with Vue projects running 2, 3, and 4 simultaneous Unity instances using the developed library.

The results for multiple instances were as follows:

- 2 instances: Page loading times were identical to the single instance case, but memory usage increased to a maximum of 30MB (as shown in the Figure 3.2), and the frame rate dropped to 95 fps. Network loading time nearly tripled to 3490 ms due to loading duplicate Unity build files.
- 3 instances: Page loading times remained similar to the single-instance case. Memory usage reached almost 40MB (as shown in the Figure 3.2), and the frame rate dropped to 60 fps. Network loading time increased slightly to 3.7 seconds.

- 4 instances: Page loading times were similar to the single instance case. Memory usage was slightly higher, nearing 50 MB (as shown in the Figure 3.2), and the frame rate dropped to 45 fps. Network loading time was 6.8 seconds.

Performance was observed to decrease linearly with respect to the number of instances. This is due to the resources being shared across multiple instances. Not only is this trend expected, but it also demonstrates that the library in use does not introduce significant overhead when handling multiple instances.

3.6.5 Conclusion of General Testing

Overall, the integration of Unity into a Vue project using the developed library did not introduce significant performance penalties, particularly when a single Unity instance was used. However, managing multiple Unity instances concurrently presented challenges related to memory usage and framerate, with performance decreasing linearly as the number of instances increased. Nonetheless, this is to be expected given that the same resources are shared by several instances.

3.7 Message Throughput Testing

The objective of this test is to determine the maximum number of Unity method calls that could be initiated from JavaScript (Vue) per second without negatively affecting the game's performance.

The methodology involved repeatedly calling a specific Unity method for a duration of 5 seconds. The chosen function moved a cube within the scene, representing an action that performs an actual change in the simulation and is thus representative of functions used in a real application. The average number of calls per second was then calculated.

The tests compared the throughput in the standalone Unity HTML Build, the Minimal Vue project, and the Realistic Vue project with one instance. Message throughput was also evaluated for multiple instances.

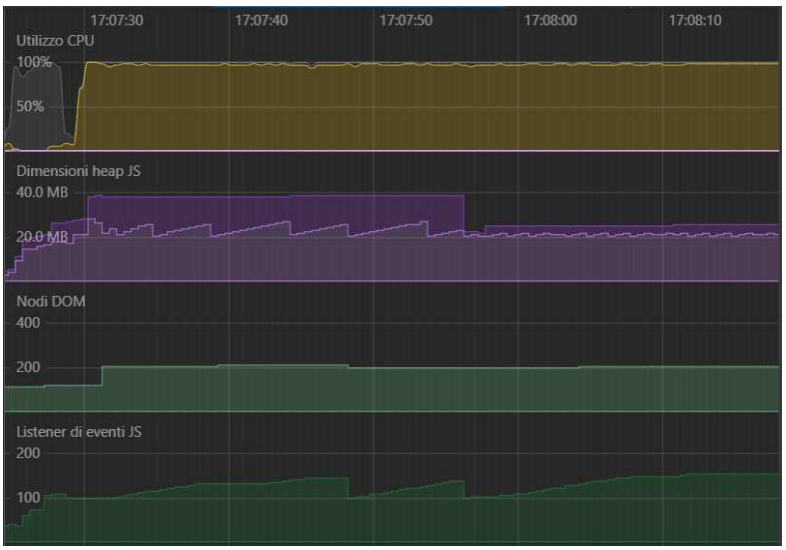
Results

The average numbers of calls per second observed for one instance scenarios are indicated in the Table 3.1

For multiple instances, the average calls per second are reported in the Table 3.2.

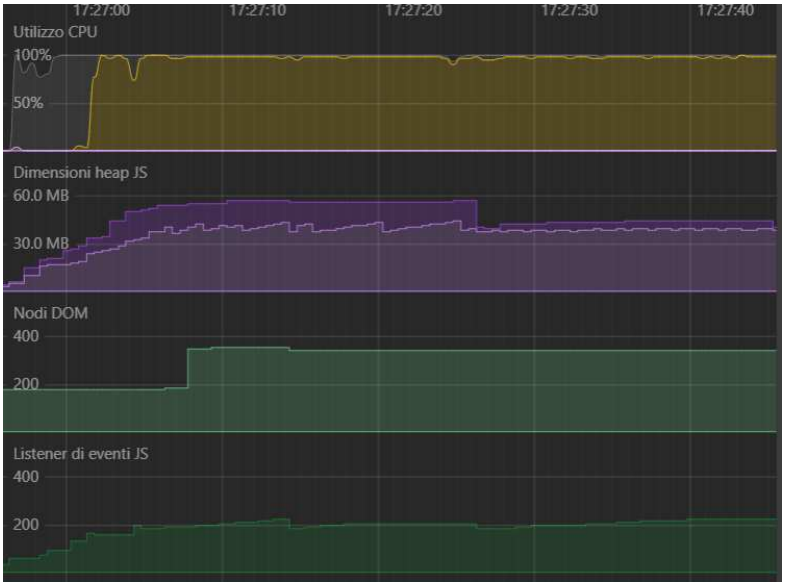
The tests indicated that the library could handle an average of approximately 140 calls per second without causing the Unity instance to freeze. The results suggest that the Vue library does not introduce significant overhead in this process compared to Unity's default implementation. In scenarios with multiple instances,

the total number of calls is distributed among the instances. Even with more than three instances, no major performance drop related to message handling was observed within the individual instance capacity.



(a) 2 instances in the same page

(b) 3 instances in the same page



(c) 4 instances in the same page

Figure 3.2: Performances recorded using the Microsoft Edge's performance tool on pages that integrate: (a) 2, (b) 3, (c) 4, instances in the same page. This image highlights that memory increases linearly as the number of instances increases.

Run Type	**Average Calls/sec**
Unity HTML Build	137.15
Minimal Vue	144.25
Realistic Vue (1 instance)	143.55

Table 3.1: Performance comparison of different run types

Instances	**Average Calls/sec**
2	82.4
3	35.93
4	32.65

Table 3.2: Average Calls per Second by Number of Instances

3.7.1 Event Listener Implementation Testing

This test aimed to evaluate the execution time and memory usage associated with different implementations for handling event listeners in the `GlobalChannel`. This object manages the event listeners registered in Vue and emitted by Unity. Different implementations for this listener mechanism were tested:

1. Direct calls (*Direct*): Unity directly invokes a method defined in Vue via `GlobalChannel.method()`. This implementation does not support executing multiple callbacks for a single Unity event.

2. Arrays of callback (*Array*): Allows registering multiple callbacks for a specific Unity event. Callbacks are stored in an array and executed sequentially when Unity publishes an action.

3. Map of callback (*Map*): Similarly to the *Array* implementation, but utilizes a Map data structure to store event-callback associations instead of an array of callbacks.

These implementations were evaluated in two scenarios: a single listener and multiple listeners (specifically, 100 listeners tested with the Map and Array implementations). The metrics collected were the execution time for the publication of events and the memory usage of the `UnityGlobalChannel`.

Results

The results for average execution time and memory usage are listed in the Table 3.3.

Single Listener			
Listener Type	**Listeners**	**Average Execution Time**	**Memory Usage**
Map	Single	0.57 ms	708 B
Array	Single	0.66 ms	832 B
Direct	Single	0.59 ms	596 B
Multiple Listeners			
Listener Type	**Listeners**	**Average Execution Time**	**Memory Usage**
Map	100	23.01 ms	8.64 KB
Array	100	26.11 ms	9.48 KB
Direct	100	Not Supported	Not Supported

Table 3.3: Execution Time and Memory Usage by Listener Type and Scenario

For scenarios requiring the handling of multiple listeners, the *Map* implementation demonstrated the most efficient performance, exhibiting minimal overhead in terms of both execution time and memory usage. In single listener scenarios, the Map implementation performed comparably to the *Direct* implementation while requiring only a marginal increase in memory (approximately 100 B).

The *Map* implementation for event listeners was identified as the most efficient approach for handling events from Unity, as it can handle scenarios with multiple listeners while adding a moderate overhead compared to the *Direct* implementation.

3.8 Conclusion

The developed library provides a convenient solution for integrating Unity simulations into Vue.js applications. By ensuring bidirectional communication and lifecycle management, the library allows Unity to be used as a rendering engine within web-based systems where front-end logic remains under Vue's control.

In addiction, based on the performance evaluation, the integration of a Unity WebGL build with a Vue project using the developed library performs comparably to Unity's native HTML build when operating with a single instance. The library also successfully enables the possibility of implementing multiple instances on the same page. Although managing concurrent instances introduces challenges related to memory consumption and frame rate, the library does not introduce a considerable overhead; in fact, performance decreases linearly as expected due to resource sharing. Furthermore, the library facilitates efficient communication between the Vue application and the Unity instance, handling a high volume of messages per second without causing performance degradation.

Chapter 4

3D LiDAR Reconstruction

The use of LiDAR sensors in the aerospace domain is widespread. Applications range from distance estimation to the scanning of structures for object detection, structural diagnosis, or relative pose estimation.

A LiDAR sensor emits laser pulses and records the time it takes for the light to return after bouncing off surfaces. This process generates a point cloud, a set of spatially distributed points in three-dimensional space representing the geometry of the scanned object. While certain calculations and analyses can be performed directly on the point cloud this representation is not always optimal for visual interpretation or further processing.

Point clouds lack explicit topological information; that is, the relationships between individual points are not defined a priori. This absence of structure makes the direct visualization of point clouds complicated, especially in contexts requiring an intuitive or detailed understanding of the scanned geometry. When rendered as separate points in three-dimensional space, the perception of the underlying object can be misleading or incomplete.

To overcome these difficulties, surface reconstruction techniques are adopted to infer a continuous 3D model from the point cloud data. The goal is to estimate a geometric surface that both precisely represents the scanned object and is visually pleasant. However, surface reconstruction is an inherently ill-posed problem: multiple valid surfaces can be inferred from the same point cloud, and it becomes exponentially more complex when the data is incomplete or noisy.

Various remeshing techniques exist to address this problem. These algorithms typically operate under certain assumptions, for example, enforcing smoothness constraints or filling in missing regions. For this work, a selection of traditional remeshing methods suitable for generic 3D models was adopted. Machine learning-based approaches were intentionally excluded due to the unavailability of large, domain-specific datasets required for training. In addition, prior research [44] has shown that machine learning models trained on general-purpose datasets may

underperform in specialized contexts when compared to traditional methods.

The primary objective of the tool developed in this work is to evaluate the performance of different remeshing algorithms and identify optimal parameter configurations that yield high-quality reconstructions. To achieve this, a simulation and benchmarking framework was implemented, based on the library provided by the authors of *A Survey and Benchmark of Automatic Surface Reconstruction from Point Clouds*[44].

The development process began with the design and implementation of a LiDAR simulator capable of producing synthetic point clouds from virtual scans. A set of CAD models representing satellite structures from real space missions (provided by ALTEC and other aerospace sources) was selected to ensure the relevance and representativeness of the test cases. Subsequently, a suite of functions and procedures was developed to facilitate the systematic testing and optimization of the remeshing algorithms.

Finally, the remeshing methods that offered the best result base on the metrics used for testing were identified.

4.1 Surface Reconstruction Algorithms

In this work, a selection of classical surface reconstruction algorithms was evaluated to determine their performance in reconstructing 3D surfaces from LiDAR-generated point clouds. These algorithms were chosen for their general applicability across various scanning conditions and object geometries.

Neural network-based surface reconstruction methods typically do not outperform classical methods unless specific training is performed on domain-relevant datasets [44]. An exception is the Implicit Geometric Regularization (IGR) [45] method because training is executed with the CAD model itself, which can surpass traditional techniques but is not practical in the scenario under analysis due to its high computational requirements.

Given the lack of sufficiently large or representative datasets, this study focused exclusively on traditional reconstruction techniques. These methods, although unable to learn priors from data, offer stable and interpretable performance and do not require training.

The five surface reconstruction algorithms considered in this study are described below.

Screened Poisson Surface Reconstruction (SPSR) The Screened Poisson Surface Reconstruction algorithm [32] builds on the classical Poisson reconstruction method, generating watertight and smooth surfaces from point clouds. SPSR is particularly suitable for scenarios involving noisy or incomplete data, as it can

produce coherent and continuous surfaces even when the scan does not fully capture the geometry of the object.

Robust and Efficient Surface Reconstruction (RESR) RESR [33] formulates surface reconstruction as an energy minimization problem that explicitly models the scanning process. RESR is specifically developed to handle noisy and sparse point clouds, such as those typically produced by real-world scanning systems.

It is capable of filling missing regions and dealing with imperfections in the data by leveraging geometric consistency heuristics.

Ball Pivoting Algorithm (BPA) The Ball Pivoting Algorithm [34] constructs a triangle mesh by iteratively "rolling" a virtual ball of user-defined radius over the input point cloud. A triangle is formed when three points are touched simultaneously by the ball without including any other point inside.

The Ball Pivoting Algorithm is a simple and memory-efficient technique. BPA performs well on uniformly sampled and complete point clouds; however, it is highly sensitive to the choice of the pivoting radius. This sensitivity limits its robustness in the presence of noise, outliers, or non-uniform sampling density. As a result, BPA may struggle to produce accurate reconstructions in scenarios where the scanned object contains holes or regions with insufficient point coverage.

Advancing Front Surface Reconstruction (AFSR) AFSR [35] uses a surface-based Delaunay triangulation approach to incrementally build a mesh from the point cloud.

This methodology promotes the generation of orientable, manifold surfaces and allows the reconstruction process to adapt locally to the structure of the input data.

Scale Space Surface Reconstruction (SSSR) The SSSR method [36] reconstructs surfaces by first simplifying the point cloud through a scale-space smoothing process. At a coarser resolution, a triangulated surface is generated, and the result is then gradually mapped back to the original scale. This approach is particularly robust to noise and can handle moderate amounts of outliers.

However, it does not attempt to infer missing geometry in undersampled regions (e.g., occlusions), as it avoids speculative surface generation. SSSR excels in scenarios where data quality is reasonable but may suffer when large gaps or uneven distributions are present in the point cloud.

4.2 Design of the Evaluation Framework

The goal of this work is to develop a testing framework capable of evaluating the performance of different surface reconstruction algorithms in terms of both reconstruction accuracy and execution time. Additionally, the framework must include an automated parameter optimization algorithm to identify the best-performing configuration for a given dataset. This enables systematic comparisons across algorithms and provides reliable guidelines for future applications.

Ultimately, the insights obtained from this evaluation process are intended to inform the integration of the most effective remeshing method into ALTEC's future projects. To support this, the framework must not only produce benchmarking results but also include integration guidelines and a Docker-based deployment setup for easy reuse and reproducibility chapter 6.

To meet these objectives, the framework was designed to implement the following core functionalities:

- Automatic loading of 3D models from the input dataset,
- Generation of synthetic LiDAR scans using a virtual scanning simulator,
- Postprocessing of the scans to create point cloud of various size,
- Execution of surface reconstruction for each model using the remeshing algorithms described above and their default parameters,
- Computation of evaluation metrics for each reconstruction result,
- Parameter optimization to improve reconstruction quality,
- Re-evaluation using the optimized parameter sets.

The framework is built upon the structure introduced in the paper *A Survey and Benchmark of Automatic Surface Reconstruction from Point Clouds* [44]. While the original purpose of that benchmark was to systematically compare traditional and learning-based surface reconstruction methods on predefined datasets, its architecture provided a solid foundation for this work.

In particular, the referenced project offers utilities for integrating new remeshing algorithms and evaluating them using standardized metrics. However, the framework developed in this thesis is designed to operate on a custom dataset, rather than relying on the benchmark's predefined datasets, and on specific remesh algorithms.

To enable this flexibility, a generalized pipeline was defined. This includes:

- Standardization of the input model format (using the OFF format),

- Definition of general-purpose scripts and evaluation functions to compute relevant metrics and store the results in a reusable format.

The developed tool can be easily adapted to work with any other dataset in the future, making it reusable beyond the specific scope of this thesis.

4.3 LiDAR Simulation

The first step in evaluating remeshing algorithms involves the generation of realistic point clouds that closely resemble those acquired in real-world scenarios. Since no suitable public datasets containing point clouds of satellites objects are available, and acquiring real LiDAR scans is often impractical or costly, a custom LiDAR simulation tool has been developed for this purpose.

This virtual scanner enables the generation of synthetic point clouds from 3D models, which serve as input for the reconstruction algorithms under test. The motivations behind this choice are:

- **Dataset flexibility:** Using a simulation-based approach allows full control over the geometry and variety of the models included in the dataset. This enables testing the reconstruction algorithms on models relevant to the studied scenario without being limited by the availability of real scans.
- **Practical constraints:** Performing real LiDAR scans is often unfeasible due to hardware limitations, logistical difficulties, and the unavailability of the target object for physical scanning.
- **Evaluation consistency:** A virtual LiDAR scanner ensures that the generated point clouds remain consistent with the underlying 3D models used as ground truth. This allows accurate evaluation of the reconstruction results, since all metrics can be computed with respect to the known reference geometry, while still retaining the noise, sparsity, and occlusion patterns typical of real scans.

Other methods for generating point clouds from 3D models do exist, but the goal was to develop a customizable simulator suitable for the scenarios described in this thesis. Existing LiDAR simulators do not meet the specific requirements of this study and either require extensive modifications [46] or come with licensing costs.

There are also synthetic scanning tools that do not rely on raycasting simulations [31][47], but these require manifold, watertight meshes. Since complex meshes are rarely manifold, such tools cannot be applied to datasets used in this study.

The following subsections describe the internal logic, implementation details, and usage workflow of the developed LiDAR simulator.

4.3.1 LiDAR Simulation Functionality

The LiDAR simulator was implemented using Unity's raycasting capabilities and is designed to emulate realistic scanning conditions similar to those encountered in space missions. The simulation models a sensor that follows a circular orbital path around the target object, mimicking a fly-around acquisition trajectory. The sensor can complete multiple full orbits, with the orbital plane that can tilt after each orbit to increase coverage and realism; a scheme of this scenario is shown in the Figure 4.1. This scanning setup allows the generation of point clouds that naturally incorporate real-world defects such as occlusions and irregular sampling densities.

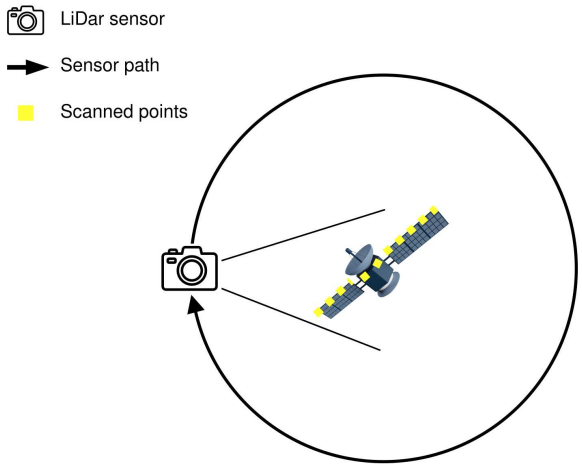


Figure 4.1: Scheme of the LiDAR simulation scenario

Point Cloud Characteristics

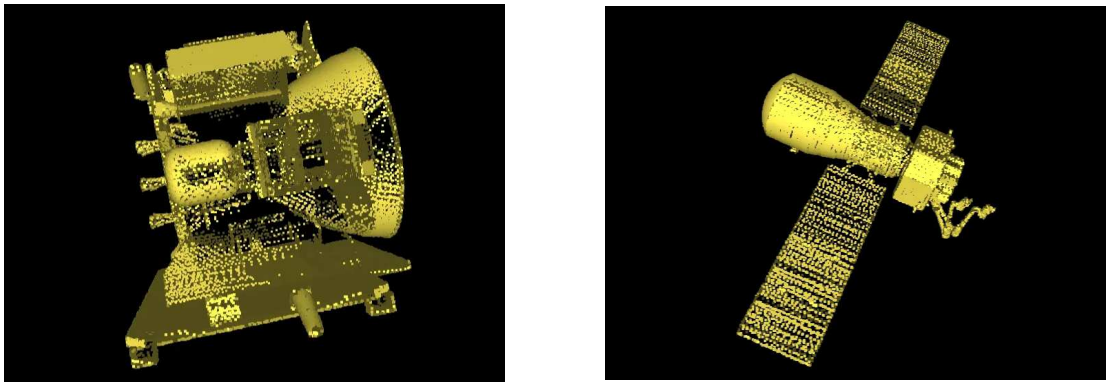
The simulated point clouds exhibit features that closely resemble those obtained from real LiDAR scans. Due to the fixed scanning trajectory and the geometry of the sensor, certain areas of the 3D model may remain occluded or only partially observed. This results in the presence of holes in the point cloud, particularly in regions consistently shadowed from the sensor's line of sight, as shown in Figure 4.2.

Moreover, the density of the sampled points varies across the model surface. Areas that remain farther from the sensor during the scan tend to be covered by fewer rays, leading to sparser point distributions Figure 4.2.

These characteristics are essential for evaluating the robustness of surface reconstruction algorithms under challenging conditions.

Scene Architecture

The simulation scene is composed of three key components:



(a) (b)

Figure 4.2: Examples of point cloud generated from the lidar simulator. (a) example of holes in point cloud, (b) example of density variation in point cloud.

- A sensor: Emits rays according to a configurable angular resolution and field of view, centered towards the scanning trajectory.
- The path: A spline defining the orbital trajectory of the sensor around the target model. After each full orbit, a tilt is applied to the path to simulate different observation angles.
- Model Loader: A mechanism that automatically loads and positions 3D models at the center of the orbit. All models are normalized to fit inside the unit cube to ensure consistent scaling across scans.

Sensor and path parameters can be adjusted via command-line arguments, making the simulation highly flexible and adaptable to different test conditions.

Workflow

Upon initialization, the simulator reads a list of 3D models to be processed. Each model is loaded individually, scaled to fit inside the unit cube, and positioned at the center of the scanning trajectory. The sensor then begins the acquisition process by performing a full circular orbit around the model.

After each session is completed, the sensor path is tilted along a defined axis by the amount given as a parameter. This operation is repeated for a predefined number of times to obtain a comprehensive scan.

During the scan, the sensor captures a series of *frames*, each representing a snapshot of rays emitted within the sensor's field of view. For every frame, the rays intersect with the model surface, generating a corresponding set of 3D points.

For each captured frame, two versions of the data are saved:

- A clean version that stores the exact 3D coordinates of the intersection points.
- A noisy version, obtained by applying the Gaussian noise model described in section 4.3.1 to the clean points.

The outputs of the simulation are the clean and noisy final merged point cloud (representing the complete scan) and all intermediate per-frame clean and noisy point sets. This structure provides traceability over the data generation process and allows for in-depth analysis not only of the final point cloud but also of all the intermediate frames. In the end, all the point clouds are saved in OFF file format in a designated folder.

By saving both clean and noisy point clouds, it is possible to evaluate not only remeshing performance based on noise levels but also the effect that different noise parameters could have on the point cloud.

Noise Model

To approximate real-world sensor behavior, a noise model is used to perturb the original position of each point in the scan. The noise model is designed to produce results similar to real scans without introducing too much complexity. The noise model is a simplification of the model described in the paper *LiDAR Data Noise Models and Methodology for Sim-to-Real Domain Generalization and Adaptation in Autonomous Driving Perception* [48]

The applied noise follows a zero-mean Gaussian distribution, where the standard deviation is a function of the distance between the sensor and the intersection point:

$$\text{noise} = \mathcal{N}(\mu, \sigma), \quad \mu = 0, \quad \sigma = \sigma_{\text{base}} + d \cdot \sigma_d$$

where σ_{base} is a fixed base noise level and σ_d is a scaling factor based on the distance d from the sensor to the target point.

The perturbation is applied to the point along the scanning direction because the majority of the error is often in the sensor's estimated distance.

In addition to the noise model, the LiDAR simulator also handles outliers. An outlier probability parameter is used to determine the likelihood that a scanned point is an outlier. When a point is classified as an outlier, it is randomly relocated within the unit cube. Although this approach does not reflect a physically accurate model of outliers, it is straightforward to implement and still provides a useful means of evaluating the robustness of remeshing algorithms in the presence of outliers.

Simulation Parameters

The simulation is designed to be highly configurable to accommodate various application scenarios. The main parameters are listed below:

- **Path radius:** Radius of the sensor's orbital path.
- **Frames per rotation:** Number of sensor frames acquired during each full rotation.
- **Number of rotations:** Total number of orbits performed by the sensor.
- **Path tilt:** Angular rotation applied to the orbital plane after each orbit is completed.
- $\boldsymbol{\sigma}_{\text{base}}$: Base standard deviation of the Gaussian noise model.
- $\boldsymbol{\sigma}_{\boldsymbol{d}}$: Noise intensity factor proportional to the point-to-sensor distance.
- **Outliers:** Probability of injecting outlier points in the scan.
- **Field of view (FOV):** Angular range of the sensor's horizontal and vertical scanning window.
- **X resolution / Y resolution:** Number of rays emitted in horizontal and vertical directions, respectively.

The total number of rays per frame is given by the product of X and Y resolution. Rays are uniformly distributed across the solid angle defined by the sensor's FOV.

4.3.2 Implementation Details

The LiDAR simulator has been developed using the Unity engine, leveraging its physics system and ray casting functionality. The core idea behind the implementation is to produce a standalone executable that can be launched from the command line, allowing automated and configurable scan sessions.

Command-Line Interface

To support headless execution and parameter configuration via CLI, the simulator integrates the `UnityCommandLineParser` library. This component handles the parsing of command-line arguments, enabling users to define key parameters such as the simulation settings, input model directory, and output data path.

Models to be scanned are specified in a list file (`.lst`), located in the input directory. This file enumerates the names of OBJ-format 3D models to be processed. At runtime, the simulator reads the list and sequentially loads each model.

Model Loading and Scene Preparation

The loading process uses a custom `ObjImporter` component, which imports each model, normalizes it to fit inside the unit cube, and positions it at the center of the scene. As the models are loaded at runtime, each imported object (and its sub-objects) is dynamically assigned a `MeshCollider`. This component is essential for enabling ray-based intersection detection, as it allows Unity's physics system to compute accurate ray-mesh collisions.

Sensor Motion and Animation

The movement of the LiDAR sensor is governed by a circular spline path constructed using Unity's spline system. To animate the sensor along this path, a `SplineAnimate` component is attached to the sensor GameObject. The speed of animation is calculated based on the *frames per rotation* parameter, ensuring that a scan is performed at each step of the path.

After completing a full orbit, the sensor path is tilted according to the user-defined rotation increment. This step simulates different observation angles.

LiDAR Scanning and Raycasting

The core of the simulation involves synthetic raycasting. A custom scanning script attached to the sensor emits rays uniformly spread along a frustum, defined by the sensor's FOV and resolution parameters.

Each ray is tested for intersection with the target model. If a hit is detected, the following data are recorded:

- The exact intersection point (clean point).
- The surface normal at the intersection.
- A noisy version of the point, obtained by applying a Gaussian noise model with parameters σ_{base} and σ_d (distance-dependent component).
- A noisy normal, distorted using the same noise logic.

In addition, each point has a probabilistic chance of being replaced by an outlier.

For visualization and debugging purposes, an arrow is rendered in the scene for each ray, pointing in the direction of the normal at the hit point.

Output and Logging

For every scan frame, both the clean and noisy point data are saved. These are organized in subfolders, enabling full traceability of each frame. At the end of the

scanning process for each model, the final complete point clouds are saved in the specified output directory.

The simulator also produces detailed logs indicating the progress of the scan session and highlighting any encountered errors, making it suitable for batch processing and integration in automated pipelines.

Automation and Batch Processing

The overall system is designed for scalability and automation. Once the `.lst` file and command-line parameters are set, the simulator autonomously loads and scans each model. This allows for the rapid generation of large datasets without manual intervention.

4.4 Evaluation Metrics

To evaluate the quality of the mesh reconstructed by the algorithms used in the framework, a set of evaluation metrics has been used. These metrics are drawn from literature and benchmark studies on 3D shape reconstruction and surface approximation [44],[49].

The aim is to quantify how closely the reconstructed mesh approximates the original reference geometry, with attention to aspects such as surface coverage, local accuracy, and continuity. The selected metrics are:

- **Intersection over Union (IoU):** This metric measures the volumetric overlap between the reconstructed mesh and the ground truth. It is computed by voxelizing both meshes and calculating the ratio between the intersection and union of the occupied voxels. A higher IoU indicates that the reconstructed shape captures the general structure and topology of the original object.

- **Chamfer Distance:** The Chamfer Distance evaluates the average bidirectional distance between the surfaces of the ground truth and the reconstructed mesh. For each point on one surface, the closest point on the other is identified, and the distances are averaged. This metric is sensitive to local surface deviations and provides a measure of geometric accuracy.

- **Normal Consistency:** This metric compares the orientation of surface normals between the two meshes. It is computed by averaging the cosine similarity of normals at corresponding points. A high normal consistency implies that the local surface smoothness and curvature are preserved in the reconstruction.

- **Execution Time:** Beyond accuracy, the computational performance of the reconstruction process is also taken into account. The total execution time includes pre-processing, reconstruction, and post-processing.

4.5 Dataset

To evaluate the performance of the reconstruction framework, a set of 3D models representing various satellite configurations were used. The primary goal is to reconstruct a mesh from synthetic LiDAR scans, assessing how well the system generalizes to different types of objects commonly found in space missions utilizing LiDAR technology.

To support this objective, a small dataset was created, composed of 3D models of different satellites with diverse geometries and structural features. These models serve to test the robustness and flexibility of the reconstruction process under varying conditions.

Since the original models may contain an excessively high number of vertices and complex mesh structures, a preprocessing step was introduced to simplify the geometry while preserving the overall shape and topology. This was achieved using Blender:

- The models are first imported into Blender, where the centroid of each mesh is computed based on its surface geometry. The mesh is then translated so that the centroid aligns with the center of the scene, ensuring consistency across different samples.
- A *voxel remeshing* modifier is applied, with parameters adjusted to produce a regularized mesh that closely approximates the original geometry.
- The resulting mesh is further simplified using a decimation algorithm, reducing the vertex count to a maximum of 100,000 vertices. This ensures that the models remain computationally manageable while retaining sufficient detail for meaningful reconstruction.

The final processed models are exported in the `OFF` format using MeshLab, making them compatible with the analysis tools used throughout the evaluation pipeline.

4.6 Implementation Details of the Testing Framework

The performance evaluation framework was developed in Python, using libraries for 3D data processing. In particular, `Open3D` and `PyMeshLab` were adopted as the core

tools for mesh manipulation and analysis. These libraries provide a comprehensive set of functionalities for point cloud handling, mesh processing, and geometric computations, making them well-suited for the requirements of this study.

The structure of the project is organized into modular components, each responsible for a specific set of tasks within the evaluation pipeline. The key elements are as follows:

- **DSR Benchmark:** The framework is built upon a modified version of the DSR benchmark, an open-source platform for 3D shape reconstruction evaluation. In this implementation, the original repository was customized to retain only the components essential for the proposed tool, primarily focusing on the computation of evaluation metrics such as Chamfer Distance, Normal Consistency, and Intersection over Union (IoU).

- **BaseDataset:** This Python class encapsulates all the functionalities required to process a full reconstruction pipeline. It manages the loading and organization of scan data, the remeshing of point clouds, the computation of evaluation metrics, and the generation of summary results for each reconstruction algorithm.

- **Externals:** A dedicated directory that contains all external libraries and tools used for the remeshing step. The remeshing libraries were modified and adapted to be executed in the context of this project.

DSR-Benchmark Integration

The evaluation module integrated into the testing framework is based on the *DSR-Benchmark*. This benchmark provides a standardized set of tools and methods for assessing the quality of 3D mesh reconstructions, particularly focusing on geometric accuracy and surface consistency.

In the context of this project, only the components related to metric evaluation were retained and adapted. These components form the core of the quality assessment pipeline and are responsible for computing the evaluation metrics.

The fundamental concept behind the metric computation is that these quantities cannot be directly calculated from the continuous surface of 3D models. For instance, computing the volumetric overlap (IoU) between two meshes or evaluating distances (Chamfer Distance) and normal orientation similarities (Normal Consistency) requires a discrete representation. To address this, each mesh is first sampled to generate a set of representative points over its volume or surface. These sampled points are then used as a basis for metric computation.

The IoU is computed through volumetric sampling. First, the axis-aligned bounding box (AABB) of the mesh is calculated. A uniform grid of sample points

is generated within this volume. Then, using the `check_mesh_contains` function from the `libigl` library, points that lie outside the mesh are discarded. This operation is performed independently for both the ground truth mesh and the reconstructed mesh. The remaining valid points are treated as voxels, and the IoU is computed as the ratio between the size of the intersection and the size of the union of these two voxel sets.

For the Chamfer Distance and Normal Consistency, uniform sampling is performed across the surface of both meshes. For each point in one set, the nearest neighbor in the other set is found. The Chamfer Distance is then computed as the average of the squared distances between these closest point pairs, while the Normal Consistency is calculated as the average cosine similarity between the normals of the corresponding points.

The implementation relies primarily on the `MeshEvaluator` class and its `eval()` method. This method expects as input a structured dictionary containing paths to:

- The original ground truth mesh.
- The mesh produced by the reconstruction algorithm.
- A pre-sampled version of the original mesh (to ensure consistency across remeshing methods).

An example usage of this method is the following:

```
mesh_evaluator.eval(self.model_dicts, outpath=path, method="SPSR")
```

Upon execution, the method loads the pre-sampled ground truth mesh and performs a new sampling on the mesh to be evaluated. It then computes the aforementioned metrics and saves the results in CSV format. This output includes a row for each evaluated model, with the corresponding metric values and method name, allowing for easy comparison and aggregation of results.

BaseDataset Class

The `BaseDataset` class represents the core component of the evaluation framework and is responsible for managing the entire testing pipeline. It is designed to be dataset-agnostic and is initialized by specifying the path to the dataset directory.

A key design principle of this class is to store as many intermediate results as possible, reducing the need for redundant computations across multiple runs. This significantly accelerates repeated evaluations and facilitates testing under different configurations.

The class also maintains all the necessary references to external tools and remeshing methods. These are encapsulated within the class to allow seamless invocation of each remeshing algorithm during the evaluation stages.

The main responsibilities of the `BaseDataset` class include:

Data Structure Initialization Upon instantiation, it creates a data object that stores all relevant paths to the models, intermediate files, and output directories. This structured approach simplifies subsequent access to the data and ensures consistency across the evaluation steps.

Model Preparation for Evaluation The method `make_eval()` is responsible for preparing each model for metric evaluation. It performs point sampling on the ground truth meshes and stores the resulting point clouds in a temporary directory. These sampled versions are later used in metric computation to ensure consistency across remeshing methods. Additionally, the method `_scale()` is employed to normalize all models, scaling them to fit within a unit cube.

Model Scanning via LiDAR Simulation The class also provides the `scan_lidar()` method, which handles synthetic LiDAR scanning of each model using the simulation parameters described in Section 4.3.1. The method produces one or more point clouds per model by emulating a scanning trajectory and sensor behavior. It also applies a set of post-processing steps to generate multiple variants of the point cloud. These variants are modified to reflect different reconstruction scenarios and are further detailed in the testing methodology section.

Remeshing and Metric Evaluation For each remeshing algorithm integrated into the framework, the `BaseDataset` class includes dedicated methods to:

- Execute the reconstruction algorithm on the point cloud data.
- Evaluate the resulting mesh using the metrics described above.
- Perform grid search over hyperparameters to identify optimal settings for each method.

Externals

The `externals` directory contains all third-party libraries required to execute the mesh reconstruction algorithms. Each library has been integrated into the project with minimal modifications to ensure compatibility and correct execution within the evaluation pipeline.

Where necessary, deprecated functions have been replaced or updated to align with current versions of the dependencies. Additionally, a dedicated installation procedure has been defined for each library, along with a validation script that runs test reconstructions on pre-saved sample models. This script serves to verify the correct installation and functionality of all algorithms prior to execution within the main framework.

The external libraries included in the project are:

- **RESR**: `mesh-tool` is a C++ library that provides a collection of mesh manipulation scripts, including an implementation of the RESR algorithm. This library relies on CGAL (Computational Geometry Algorithms Library) as a core dependency. The original repository has been slightly adapted to ensure compatibility with the new version of CGAL.
- **SPSR**: The official implementation provided by the authors of the reference paper is used. A custom `Makefile` has been added to facilitate building the project on Linux systems, allowing it to be executed as a command-line tool within the pipeline.
- **BPA**: This algorithm is natively included in the `Open3D` library. It is directly accessed from within the Python codebase using the provided API functions.
- **AFSR and SSSR**: These methods are part of the C++ CGAL library. To integrate them, a dedicated C++ wrapper script was developed. This script accepts the necessary remeshing parameters via the command line, executes the reconstruction process, and exports the resulting mesh. The script is invoked from Python using standard subprocess calls, making the integration both modular and transparent.

All remeshing tools are accessible either natively from Python or via command-line interface (CLI) wrappers.

Testing Scripts

The previously described framework components provide the foundation for the implementation of two core testing scripts: one focused on performance evaluation of the reconstructed models and the other dedicated to measuring execution time.

The first script is responsible for evaluating the quality of the reconstructed meshes before and after parameter optimization. Initially, it performs a baseline evaluation using default parameter values for each remeshing algorithm, applied to point clouds with varying characteristics (described in detail in the testing section). Subsequently, a grid search is executed to identify the optimal set of parameters for each method, after which the evaluation is repeated to assess performance improvements.

The grid search procedure systematically explores a predefined range of values for the remeshing algorithm parameters, aiming to identify the configuration that yields the best reconstruction results. The goodness of a mesh is defined by a value that could be a composition of the before-described metrics or a single one of them. Although an initial attempt was made to guide the optimization using

a composite score that combines all evaluation metrics, this approach presented several challenges.

The proposed composite metric was defined as follows:

$$\text{Score} = \text{IoU} \cdot w_1 + \frac{1}{\text{ChamferDistance} + 1} \cdot w_2 + \text{NormalConsistency} \cdot w_3$$

$$w_1 = 0.4, \quad w_2 = 0.4, \quad w_3 = 0.2$$

The weights w_1, w_2, and w_3 were arbitrarily chosen to emphasize the importance of the Intersection over Union and Chamfer Distance metrics. However, this approach proved to be impractical. The fundamental incompatibility between the scales of the involved metrics, such as comparing normalized percentages (e.g., IoU) with absolute distances (e.g., Chamfer Distance), led to inconsistencies in the evaluation. In some cases, the composite score resulted in suboptimal configurations that degraded individual metric performance.

As a result, the final implementation relies solely on the IoU metric to guide the optimization process. This choice simplifies the evaluation and has shown more consistent improvements across the tested configurations.

4.7 Testing

The objective of this testing phase is to assess the performance of various surface reconstruction algorithms under different point cloud conditions. In particular, the experiments aim to evaluate how reconstruction quality varies depending on the density and noise level of the input data, and to identify optimal parameters for each scenario. This evaluation reflects the real-world variability of LiDAR scans and ensures the robustness and generalizability of the tested methods.

4.7.1 Experiments

To investigate the impact of different point cloud characteristics, three experiments were designed, each based on a distinct configuration of input data. The variations were introduced either by using the noisy scan or by reducing their density through post-processing.

- **Experiment 0**: High-density point cloud without noise.
- **Experiment 1**: Low-density point cloud with noise.
- **Experiment 2**: High-density point cloud with noise.

The point clouds for Experiments 0 and 2 are obtained directly from the LiDAR scan simulation, with Experiment 2 incorporating the simulated noise model. For Experiment 1, the base point cloud is further processed using the *Point Cloud Simplification* tool available in MeshLab. This tool attempts to reduce the number of points to a target value of 10,000 while preserving the overall geometry of the model as much as possible. However, the final number of points can vary depending on the specific geometric properties of the mesh and the original resolution of the scan.

Point Cloud Sizes

The point cloud sizes vary from 22000 points to 270000 for the original size and from 2000 to 21000 for the simplified ones.

The observed differences in point cloud sizes are strongly influenced by the geometry of the models. Objects with elongated structures or thin appendages typically produce sparser point clouds, whereas more compact geometries tend to yield denser scans. These variations highlight the importance of evaluating reconstruction algorithms under diverse conditions to ensure consistent performance across different structural configurations.

4.7.2 Grid Search Parameters

To determine the optimal configuration for each surface reconstruction algorithm, a grid search was performed over a predefined range of parameter values. The chosen parameters reflect those commonly used in the literature.

The following summarize the parameters explored during the optimization phase for each reconstruction method.

SPSR

Parameter	**Values**
Depth of octree	6, 8, 10, 12
Boundary conditions	Dirichlet, Neumann, Free

Table 4.1: Parameters used for SPSR during grid search.

RESR

Parameter	Values
α	16, 32, 48
σ	0.001, 0.01, 0.1, 1
λ	1.5, 2.5, 5, 10

Table 4.2: Parameters used for RESR during grid search.

BPA

For BPA, the radius of the ball used in the algorithm is derived from the average distance between points in the point cloud. This base radius is then multiplied by specific weights to produce multiple pivoting radii. The algorithm supports the use of multiple radius values to improve mesh reconstruction over surfaces with varying curvature.

Parameter	Values
Mean distance weight	1, 2, 3
1st radius weight	0.01, 0.1
2nd radius weight	0.2, 0.5
3rd radius weight	1, 2
4th radius weight	2, 5

Table 4.3: Parameters used for BPA during grid search.

AFSR (Anisotropic Feature-Sensitive Reconstruction)

Parameter	Values
Radius ratio bound	0.5, 1, 2, 4
β	0.01, 0.1, 0.5

Table 4.4: Parameters used for AFSR during grid search.

SSSR (Smooth Signed Scalar Reconstruction)

Parameter	Values
Smoothing iterations	1, 2, 3, 4
Maximum facet length	0.5, 1, 2, 5, 10

Table 4.5: Parameters used for SSSR during grid search.

4.7.3 LiDAR simulation parameters

For the validation and testing of the implemented reconstruction pipeline, a synthetic dataset was generated using the simulator described in previous sections. The simulation parameters were selected to approximate, as closely as possible, the expected characteristics of a real LiDAR sensor, within the constraints of publicly available information.

Since the precise specifications of the target sensor are not known a priori, both resolution and noise characteristics were estimated based on literature and common configurations of real sensors. In particular, no outliers were introduced during the simulation, under the assumption that the onboard processing of a real sensor would filter them out prior to point cloud generation.

The Table 4.6 summarizes the parameter values used during the test sessions.

Parameter	**Value**
Path radius	50 units
Frames per rotation	10 frames
Number of rotations	4 full rotations
Path tilt after each rotation	5°
σ_{base}	0.3%
σ_d	0.1% of distance
Outliers	0
Field of view	20°
X resolution	128 rays
Y resolution	128 rays

Table 4.6: Simulation parameters used for testing

The combination of moderate frame count and path tilting provides sufficient model coverage while still preserving realistic occlusion patterns.

4.7.4 Results

This section presents the outcomes of the testing phase for the reconstruction framework. The evaluation focuses on three main metrics: Intersection over Union (IoU), Chamfer Distance (CD), and Normal Consistency (NC). Special attention is paid to comparing the performance with default remeshing parameters versus optimized ones, obtained through a grid search.

Experiment 0

In the first experiment, the optimization process led to an overall improvement of approximately 3% across all metrics. The Ball Pivoting algorithm showed the highest benefit from parameter tuning, with a 7% increase in IoU. SPSR and RESR achieved the best average IoU values, though RESR had significantly higher Chamfer Distance. On the other hand, Ball Pivoting consistently performed better in minimizing Chamfer Distance.

Method	**IoU ↑**	**CD (*10^2) ↓**	**NC ↑**
SPSR	0.584	0.297	0.840
RESR	0.623	0.528	0.785
BPIVOT	0.461	0.270	0.870
AFSR	0.495	0.363	0.826
SSSR	0.508	0.316	0.826

Table 4.7: Mean performance values for Experiment 0 with optimized parameters

Experiment 1

In the second experiment, the optimization step had a more limited impact, leading to an average improvement of only 1.4%. Notably, SPSR showed no change after optimization. IoU values remained highest for SPSR and RESR, while Ball Pivoting again achieved the lowest Chamfer Distance. These results indicate stable performance of the reconstruction algorithms across different point cloud densities.

Method	**IoU ↑**	**CD (*10^2) ↓**	**NC ↑**
SPSR	0.561	0.397	0.820
RESR	0.610	2.190	0.746
BPIVOT	0.449	0.321	0.838
AFSR	0.446	0.435	0.815
SSSR	0.398	0.589	0.751

Table 4.8: Mean performance values for Experiment 1 with optimized parameters

Experiment 2

In the third experiment, the optimization step resulted in an average improvement of nearly 4%, with Ball Pivoting again benefiting the most (up to 10%). The outcomes closely align with those of Experiment 0, confirming the robustness of the methods to moderate noise levels.

Method	IoU ↑	CD (*10^2) ↓	NC ↑
SPSR	0.580	0.300	0.835
RESR	0.628	0.777	0.771
BPIVOT	0.457	0.516	0.842
AFSR	0.488	0.343	0.788
SSSR	0.515	0.248	0.793

Table 4.9: Mean performance values for Experiment 2 with optimized parameters

4.7.5 Qualitative Results

The Figure 4.3 illustrates the qualitative results of the five reconstruction methods tested.

Among the methods evaluated, **RESR** and **SPSR** consistently deliver the highest visual quality. These approaches exhibit fewer reconstruction artifacts such as holes and incorrect normals, and **SPSR** in particular tends to generate notably smoother mesh surfaces. On the other hand, **BPIVOT**, **AFSR**, and **SSSR** struggle to maintain normal consistency and surface continuity. The meshes produced by these algorithms are less accurate and tend to display jagged features, which diminish their overall fidelity and aesthetic quality.

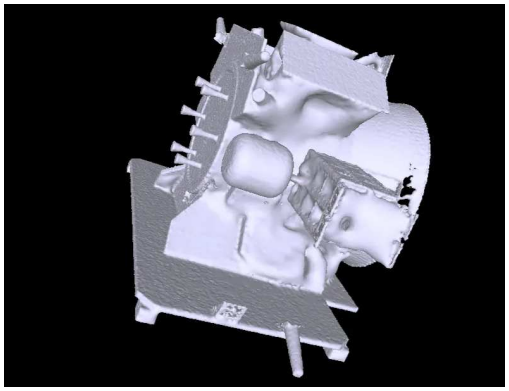
4.7.6 Execution Time

All reconstruction times include both preprocessing and post-processing stages. Benchmarks were executed on a system equipped with an Intel i7-8750H CPU, an NVIDIA GeForce RTX 2060 GPU, and 16 GB of RAM.

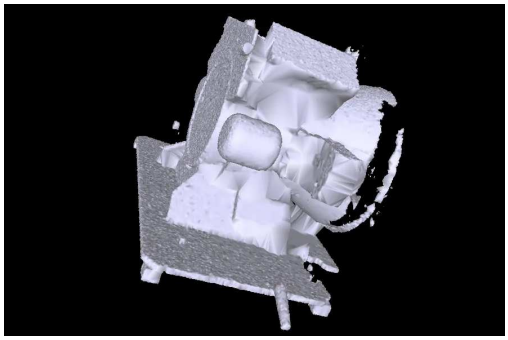
Original Point Cloud (Experiment 2) Table 4.10 presents the average execution time (in seconds) for each method when applied to the original, unmodified point cloud.

Method	Unoptimized (s)	Optimized (s)
SPSR	2.15	11.48
RESR	11.90	12.20
BPIVOT	12.47	5.42
AFSR	7.02	7.11
SSSR	14.85	8.28

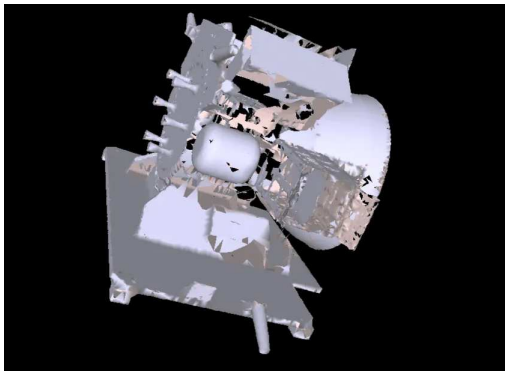
Table 4.10: Mean execution time for each method using the original point cloud (Experiment 2)



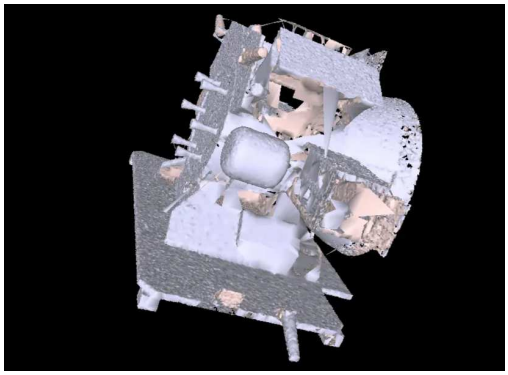
SPSR



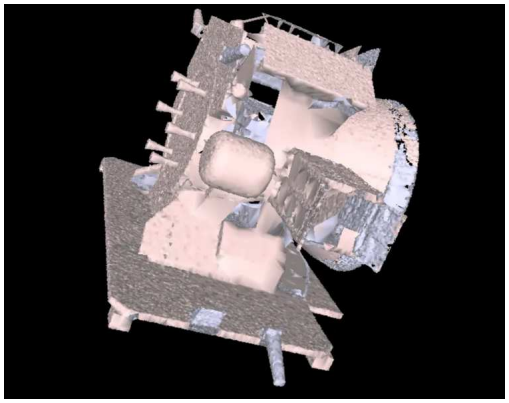
RESR



BPA



AFSR



SSSR

Figure 4.3: Meshes generated using different remeshing algorithms with the same point cloud. Different color is assigned to each face based on the normal orientation

Simplified Point Cloud (Experiment 1) Table 4.11 shows the average execution time for each algorithm when working with the simplified point cloud

dataset.

Method	Unoptimized (s)	Optimized (s)
SPSR	1.45	1.09
RESR	1.17	0.94
BPIVOT	0.34	1.16
AFSR	0.42	0.47
SSSR	3.66	1.01

Table 4.11: Mean execution time for each method using the simplified point cloud (Experiment 1)

As expected, execution times are significantly reduced when using simplified data. While SPSR and RESR remain relatively efficient even on the full-resolution dataset, methods like BPIVOT and SSSR demonstrate much greater sensitivity to input size, with SSSR being the most computationally expensive in its unoptimized form.

The application of parameter optimization reveals varied effects in the reconstruction methods. As shown in Tables 4.10 and 4.11, the variation in execution times due to optimization depends strongly on the underlying algorithm.

In particular, methods such as **BPIVOT** and **SSSR** benefit noticeably from optimization, resulting in a considerable reduction in execution time. For instance, in the case of BPIVOT, the mean time decreases from 12.47 s to 5.42 s when processing the original point cloud.

In contrast, **SPSR** demonstrates a marked increase in execution time after optimization, more than doubling its processing duration.

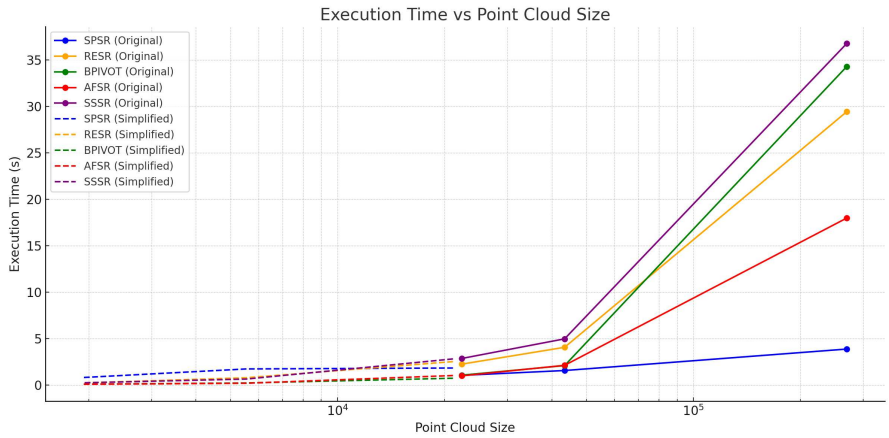
AFSR, on the other hand, shows minimal sensitivity to parameter changes, with negligible differences observed between optimized and unoptimized configurations.

Execution Time and Point Cloud Size

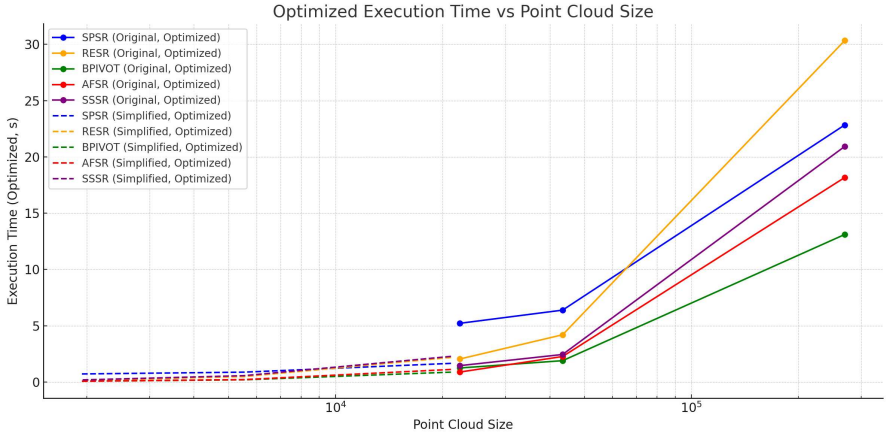
Execution time is significantly influenced by the size of the input point cloud. As expected, a larger number of points leads to increased computational load across all methods. This trend is clearly observable in Figure 4.4, where execution times grow rapidly with point cloud size, particularly for algorithms like **BPIVOT**, **RESR**, and **SSSR**.

When using simplified point clouds, the execution time decreases considerably. This reduction is evident across all methods.

Most algorithms exhibit consistent trends between optimized and unoptimized configurations, as shown in Figure 4.4. However, **SPSR** deviates from this pattern. While its unoptimized version scales linearly with point cloud size, the optimized



(a)



(b)

Figure 4.4: Execution time of the remeshing algorithms with respect of point cloud size before (a) and after (b) optimization

configuration appears to follow a more polynomial trend, aligning more closely with the behavior of the other algorithms.

However, the execution time of **SPSR** does not grow as fast as other algorithms, for example **RESR**, as the size of the point cloud grows.

4.7.7 Conclusion

The experimental results indicate that **SPSR** and **RESR** consistently deliver superior mesh reconstructions compared to other methods. These two approaches

produce smoother surfaces with fewer topological artifacts, such as holes or incorrectly oriented normals. Among them, SPSR typically generates the smoothest meshes.

Although **BPA** performs well in terms of chamfer distance, the generated meshes are often jagged and with holes. As a result, qualitatively, **BPA** performs poorly compared to **SPSR** and **RESR**.

SSSR and **AFSR** exhibit issues similar to those of BPA but also present additional problems related to surface normals. Specifically, these algorithms generate inverted normals in certain regions of the mesh compared to the original mesh.

In conclusion, **SPSR** is the best-performing algorithm overall, considering both quantitative metrics, qualitative results, and execution time. While its qualitative outputs are comparable to those of **RESR**, **SPSR** offers an advantage in terms of scalability, as its execution time increases more slowly than **RESR** with the growth of the point cloud size.

Regarding parameter optimization, the results suggest that its impact is limited. In many cases, the performance difference between optimized and unoptimized configurations is marginal. This may be attributed to the initial parameters being already well-tuned for general performance, or it could indicate the need for a more extensive parameter search to uncover potential improvements.

Chapter 5

Structural defect visualization

In the context of 3D reconstruction from real-world objects, it is often helpful to add to the resulting models additional visual information that facilitates analysis and interpretation. In addition to geometric reconstruction of the scanned object, the ability to visually highlight structural inconsistencies or deviations can be valuable, particularly in applications related to space missions.

A representative scenario is the maintenance of satellites. In such cases, the ability to visually verify the structural integrity of components by identifying missing or extra parts can support maintenance procedures or anomaly detection. For instance, during autonomous inspection, it may be crucial to determine whether all expected elements of a satellite are present and undamaged, at least from a surface-level analysis.

To support such tasks, this work explores a visualization approach that enables the detection of structural defects through a comparative analysis between two 3D models: a reference CAD model and a mesh obtained from 3D scanning of the physical object. The reference model represents the expected geometry of the object, often available in aerospace applications due to the necessity of accurate digital designs during manufacturing. The scanned model, instead, captures the actual state of the object, potentially revealing discrepancies caused by physical damage or structural wear.

The core idea is to compute the geometric differences between the scanned and reference meshes and to encode such differences as visual cues directly on the 3D surface. This is achieved by highlighting areas that deviate excessively using a color-coded scheme. As a result, the operator or analyst can immediately identify potentially problematic areas that may require further inspection.

It is important to acknowledge that this comparison is sensitive not only to

real structural defects but also to reconstruction inaccuracies, such as remeshing artifacts or partial occlusions during scanning. These limitations must be considered when interpreting the results of the visualization.

5.1 Requirements

The tool must be compatible with a web-based environment and adhere to performance and usability considerations that are critical in this context.

5.1.1 Web-Based Execution

A fundamental requirement is that the tool must be executable directly within a web browser. This constraint arises as the tool will be used inside Unity and built in WebGL to be included in a Vue project. Consequently, all client-side operations must be implemented in a way that ensures compatibility with the Unity WebGL runtime and respects the performance limitations of in-browser execution.

5.1.2 Client-Side Responsiveness

Given the interactive nature of the visualization, client-side operations must be computationally lightweight and capable of delivering a responsive user experience.

5.1.3 Server-Side Preprocessing

While real-time operations on the client are constrained, more computationally intensive tasks can be delegated to the server side. The system must support the execution of preprocessing routines on the server, such as computing geometric differences between models or generating auxiliary data structures required for visualization. These preprocessing tasks may take several minutes to complete but should be executed only once per model pair. The resulting data must be stored and retrievable, allowing subsequent client sessions to load precomputed results without repeating the entire processing pipeline.

5.1.4 Model Independence

The tool must be designed to be model-agnostic, with support for dynamically loading different mesh pairs at runtime. This includes both the scanned model and the reference model, which may vary across different use cases. The system should therefore provide mechanisms for importing arbitrary models and managing associated preprocessing results.

5.2 Visualization Method

To effectively convey the differences between the scanned mesh and the reference model, a heatmap-based visualization strategy was adopted. This approach encodes the deviation information using a color scale directly mapped onto the surface of the scanned model. Regions exhibiting significant geometric discrepancies are highlighted using different colors. This form of visual encoding provides immediate perceptual feedback and enables rapid identification of potentially defective or altered regions.

The colored scanned mesh can be combined with a semi-transparent rendering of the reference model. This layered visualization allows a direct spatial comparison between the expected and observed geometries, allowing the user to better contextualize anomalies and understand their structural implications. The use of transparency ensures that both meshes remain visible without occlusion, supporting a more comprehensive inspection.

Alternative visualization and processing techniques were considered but ultimately deemed less effective in this context. For example:

- **Side-by-side rendering:** Presenting the reference and scanned models in separate viewports allows for isolated inspection but lacks spatial correlation, making it harder to assess localized deviations.

- **Difference meshes:** Visualizing the direct subtraction between the two geometries may highlight volumetric changes, but it often results in fragmented or non-intuitive structures that are difficult to interpret without further post-processing.

- **Volumetric representation:** Using volumetric rendering to render the two models. By overposing the models and using a volumetric rendering, different colors could be assigned to volumes between deviated surfaces. However, this solution could be computationally intensive and require complex preprocessing.

- **2D descriptors**: An alternative strategy involves computing the differences between the two models using local visual features such as SIFT descriptors [50]. This method consists of rendering both the scanned and reference meshes ensuring the presence of sufficient visual features, and then extracting descriptors from the resulting images. The differences between the descriptors can be interpreted as a proxy for geometric discrepancies. One advantage of this approach is the invariance of such descriptors to transformations, potentially offering robustness to small misalignment between the two meshes. However, this method also presents several limitations. First, since descriptors operate in image space, they are unable to retain a meaningful correspondence

with the 3D geometry. For this reason and because the descriptors depend on the specific rendered view, it is not possible to precompute the SIFTs. This introduces performance challenges, especially in web-based environments with limited computational resources, as the computation must be executed on each frame. Additionally, to be viable in interactive applications, descriptor extraction and comparison must be implemented in Unity shaders compatible with WebGL, a constraint that could result to a complex implementation. Finally, as the matching is performed in 2D, the estimated discrepancies may lack geometric precision, particularly in the presence of depth ambiguities or occlusions.

In contrast, the heatmap overlay computed based on the distances between sampled points in each model offers a balance between clarity, interpretability, and rendering efficiency. It allows users to intuitively understand where deviations occur and how severe they are.

5.3 Processing workflow

To render the heatmap as described in previous sections, the visualization tool follows the steps described below:

5.3.1 Mesh Alignment

The first step consists of aligning the two meshes as accurately as possible. Since the reference model (typically a CAD mesh) and the scanned model are generated through different processes, their origins and coordinate systems generally do not coincide. Even using common heuristics such as aligning the center of mass or the geometric centroid may not suffice, especially in the presence of partial scans, occlusions, or inherent discrepancies between the real object and its virtual model.

Moreover, orientation differences must be addressed: the pose of the scanned mesh depends on the relative position between the scanner and the object, which cannot be determined a priori. Consequently, an alignment procedure is required; the Iterative Closest Point (ICP) is used as a co-registration method.

5.3.2 Mesh Sampling

Since direct surface-to-surface comparison is computationally impractical, a discrete sampling of both meshes is required. There are two primary strategies:

- **Uniform surface sampling**, where points are distributed evenly across the mesh surface, regardless of vertex distribution.

- **Vertex-based sampling**, where the existing mesh vertices are used as sample points.

While vertex-based sampling is computationally cheaper, it may result in uneven spatial coverage, especially in meshes with irregular face sizes. Uniform sampling, on the other hand, ensures more accurate distance measurements at the cost of additional preprocessing.

5.3.3 Distance Computation

Once a set of sample points has been generated, the next step is to compute the distance from each point on the scanned mesh to its nearest neighbor on the reference mesh. However, this computation alone yields unsigned distances: it is not possible to determine whether a point is located outside or inside relative to the reference surface.

To obtain a signed distance, surface normals can be employed. For each sample point p on the scanned mesh, let q be the nearest point on the reference mesh and $\vec{v} = q - p$ the displacement vector. If $\vec{v} \cdot \vec{n}\_p < 0$, where $\vec{n}\_p$ is the normal at p, then p is considered to be inside the reference surface, and the distance is assigned a negative sign.

To fully capture discrepancies in both directions, a symmetric operation is performed by computing distances from points on the reference mesh to the scanned mesh. This bidirectional comparison follows the approach used in Chamfer Distance calculations and ensures that deviations such as protrusions or missing parts are properly detected.

5.3.4 Distance-to-Color Mapping

The computed distances are mapped to colors using a predefined look-up table (LUT). This LUT assigns colors to distance ranges (thresholds), with interpolation applied for intermediate values. The resulting color encoding creates a visual heatmap of geometric deviation across the model's surface.

5.3.5 Texture Generation and Projection

To render the heatmap as a surface texture, each sampled point must be mapped to 2D UV coordinates. This process involves either UV unwrapping or UV projection of the 3D mesh. Once the UV coordinates of each point are computed, the corresponding pixel in the texture is colored based on the point's deviation value.

As samples may not cover the texture uniformly, a scattered set of colored pixels is initially obtained. These are then interpolated to produce a continuous texture,

filling gaps with the color of the nearest valid point. This results in a smooth texture that reflects the spatial distribution of geometric errors.

5.3.6 Visualization and Overlay

Finally, the texture is rendered onto the scanned 3D model using shading techniques. Visual cues such as lighting and shadows are preserved to maintain depth perception and enhance spatial understanding. The heatmap rendering is also designed to support transparency and overlay: the reference model can be rendered semitransparent, allowing users to visually compare both shapes in a single integrated view.

This visualization mode facilitates the rapid identification of structural anomalies, missing components, or unexpected protrusions, which is particularly relevant in the context of space missions where surface integrity is critical.

5.4 Implementation Details

The implementation of the visualization tool is divided into two main components: a back-end preprocessing module and a front-end rendering module. This allow the separation of computationally intensive operations from interactive rendering.

The back-end module, developed in Python, is responsible for computing the geometric differences between the reference and scanned meshes. It performs mesh alignment, surface sampling, and distance calculation as described in the previous sections. Based on a predefined color mapping scheme, the module produces a sparse texture that encodes the computed distances in color form.

The front-end component is implemented in Unity, designed to be compatible in WebGL. This module dynamically loads both 3D models and the precomputed texture from the server. It also performs texture completion, filling in the gaps in the sparse texture to create a continuous heatmap, and handles the visual rendering of the model with appropriate shading and lighting effects.

The following subsections provide further details on the implementation of both components.

5.4.1 Back-End Preprocessing Module

The back-end preprocessing component is implemented as a Python script, designed to be executed from the command line and parametrized via console arguments. It is responsible for the core geometric computations and for generating the heatmap texture used by the visualization tool. The script is modular and optimized to reduce redundant computations through intermediate result caching.

At a high level, the script follows the steps listed below:

1. **Mesh loading and simplification:** The source (scanned) and target (reference) meshes are loaded from disk. Optionally, mesh simplification can be applied to reduce computational load during subsequent processing.
2. **UV mapping:** UV coordinates are generated for both meshes.
3. **Mesh sampling:** Points are sampled from both meshes. Uniform surface sampling is used. Sampled points are stored in cache to avoid re-sampling.
4. **Optional alignment (ICP):** If enabled, an Iterative Closest Point (ICP) algorithm is applied to align the source mesh to the target mesh. This step addresses possible discrepancies in translation and rotation between the two models.
5. **Distance computation:** Point-to-surface distances are computed in both directions (source-to-target and target-to-source), effectively implementing a bidirectional Chamfer distance. Computed distances are cached for reuse.
6. **Normalization and color mapping:** The raw distances are normalized according to a specified scale and mapped to colors using a breakpoint system. This results in a color-coded representation of deviations. The result of this step is shown in the Figure 5.1
7. **Texture generation:** Colored distance values are projected onto the UV space, resulting in a sparse texture (dotted texture) encoding the colorized distances. The outcome is displayed in the Figure 5.2

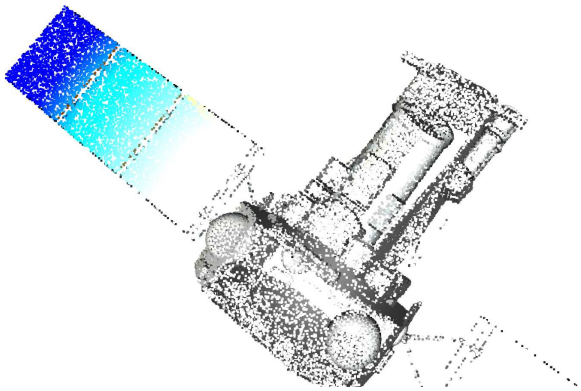


Figure 5.1: Colored point cloud resulted from the sampling, distance calculation and colorization steps. Points with blue color indicate missing parts

To optimize execution time and enable incremental workflows, all intermediate results are saved as cache files. When using the same mesh configuration, the

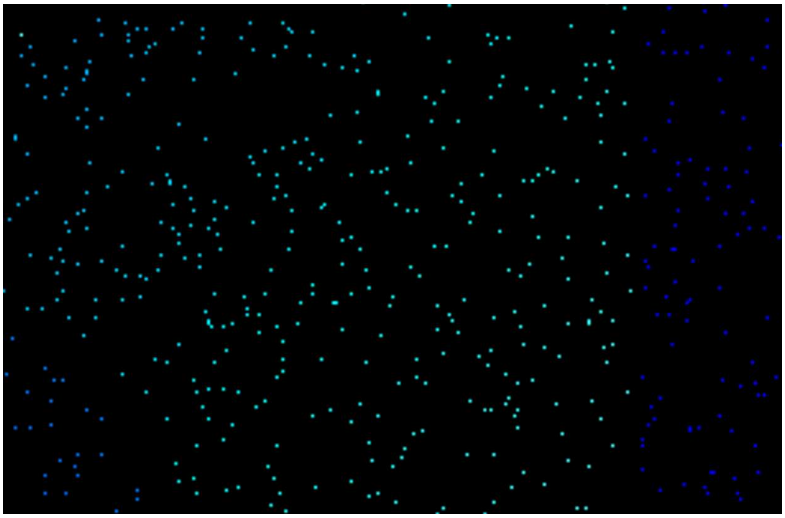


Figure 5.2: Dotted texture resulting from the point to texture mapping stage

script avoids repeating steps such as sampling and distance computation. Proper management of the output and cache directories is essential to ensure the caching mechanism works as intended; different mesh pairs must be processed using separate output paths.

The script produces the following output files in the specified output directory:

- `source.obj`: The processed source mesh with UV mapping.
- `target.obj`: The processed target mesh with UV mapping.
- `points.png`: A dotted texture encoding color-mapped distance values.
- `positions.png`: A dotted black-and-white texture encoding sample positions.
- `info.json`: A JSON file containing additional information.

In addition, the following cache files are stored:

- `sampled_source.npz`: Sampled data from the source mesh.
- `sampled_target.npz`: Sampled data from the target mesh.
- `distances.npz`: Computed distances between sampled points.

5.4.2 Software Structure

The implementation is divided into a main script and two supporting classes:

- `MeshTools`: This class encapsulates functionalities related to loading, transforming, and managing 3D mesh data. It handles operations such as simplification and UV unwrapping.

- `SampledMesh`: This class manages the sampled point cloud derived from the mesh surfaces. It is responsible for performing the sampling operation (e.g., uniform surface sampling), storing the resulting points, computing point-to-surface distances with respect to another mesh, and associating normal vectors when needed for sign estimation. It also provides caching functionalities to store and reload previously sampled data.

The main parses input arguments, initializes instances of the above classes, and handles the sequence of operations required to preprocess the input data.

Model Load and Simplification

The first step in the preprocessing pipeline involves loading and simplifying the two meshes. This operation is performed using the `Open3D` library. Upon loading the mesh, a simplification step is applied via the `simplify_quadratic_decimation` function, which reduces the number of vertices to a maximum of 100,000. This threshold was chosen to find a balance between preserving sufficient geometric detail and maintaining computational efficiency during subsequent processing.

Following simplification, the mesh is unwrapped to generate UV coordinates. This operation is performed using a function provided by the `pymeshlab` library, which applies a "trivial per-triangle" UV mapping strategy. In this scheme, each triangular face of the mesh is independently mapped to a corresponding triangle in UV space. These triangles are arranged in a grid-like pattern, each separated by a fixed spacing of 3 pixels to prevent texture bleeding during rasterization.

The size of each UV triangle is proportional to the size of the corresponding 3D face, controlled by a stepwise scaling mechanism that categorizes faces into size classes. This ensures that larger faces receive proportionally more texture resolution, while smaller faces are compacted accordingly, as illustrated in the Figure 5.3.

Although this trivial unwrapping method is computationally efficient and straightforward to implement, it introduces certain limitations. Notably, since no geometric projection or distortion minimization is performed, the resulting UV layout may lead to texture stretching on irregular or highly distorted faces. However, in the context of this application, such distortions have limited impact

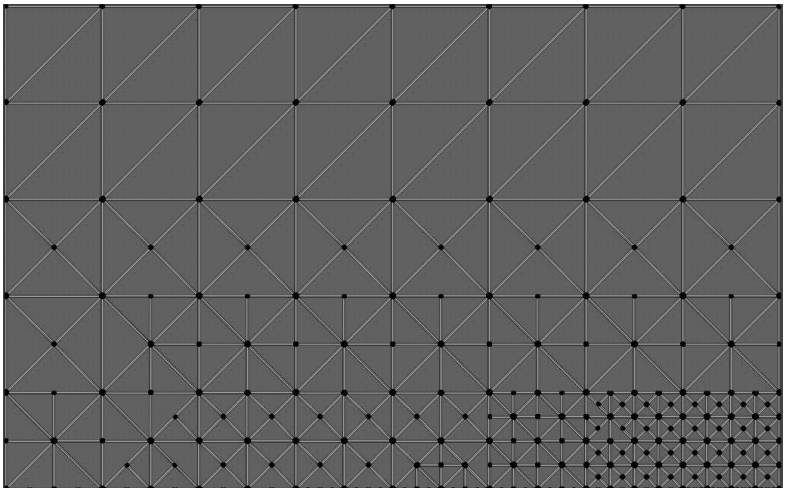


Figure 5.3: Example of trivial per-triangle UV parametrization of a complex 3D model

on the final visualization, where the emphasis lies on detecting the presence and location of surface deviations rather than rendering photorealistic textures.

The processed mesh with its associated UV coordinates is saved to disk in the cache folder.

The same workflow is computed for both the source and target mesh.

Sampling

Once the mesh is loaded and UV-mapped, it is converted into a `SampledMesh` instance by `SampledMesh.from_mesh(mesh, num_samples)` method. This step performs uniform sampling of the mesh surface, where the total number of sampled points is defined as a configurable parameter.

The adopted sampling strategy ensures that the distribution of points is proportional to the surface area of each face, thereby capturing geometric detail uniformly across the model. Specifically, the area of each triangle is first computed, and the number of points assigned to each face is determined in proportion to its relative area with respect to the entire mesh. To guarantee full coverage, even the smallest faces are assigned at least one sample point. For these cases, a single point is deterministically placed at the centroid of the face to make downstream processing work better, in particular the texture filling operation performed on the client side.

Uniform sampling over triangular faces is implemented using barycentric coordinates. For each point to be sampled, three random barycentric coefficients (u, v, w) are generated such that $u + v + w = 1$ and $u, v, w \geq 0$. These values are then used to interpolate a position within the triangle, yielding a uniformly distributed point

over its surface.

An attempt was made to parallelize the sampling procedure using multithreading to improve performance. However, due to the overhead associated with Python's thread management, this optimization did not provide significant gains.

Nonetheless, the current implementation remains performant, largely thanks to efficient use of vectorized operations provided by the `NumPy` library.

At the end of the operation, the `SampledMesh` class retains the sampled points, their corresponding surface normals, and the index of the face from which each point was generated. This data is later used in the point-to-texture mapping stage to correctly associate sampled data with surface topology.

ICP Registration

If enabled via configuration parameters, an additional alignment step is performed using the Iterative Closest Point (ICP) algorithm. This operation aims to reduce misalignments between the scanned mesh and the reference mesh, which may result from differences in acquisition pose, scale, or local geometric inconsistencies.

The registration process is carried out using the point clouds obtained through the sampling procedure described earlier. Specifically, the centroids of both the source and target point clouds are computed and used to apply an initial coarse alignment based on translation. This translation brings the two point sets roughly into the same coordinate frame before the ICP refinement.

Subsequently, the fine registration is performed using the `registration_icp` function from the `Open3D` library, configured with the `TransformationEstimationPointToPlane` mode. This method minimizes the point-to-plane distances between the two point clouds.

Once the optimal rigid transformation is estimated, the computed translation and rotation are applied back to the original triangle meshes. In this way, the alignment step influences not only the sampled point clouds but also the source and target mesh geometry.

Distance Computation

Once the source and target meshes are aligned and sampled, the next step in the preprocessing pipeline consists in computing the distances between the two surfaces to identify geometrical deviations. This is done in two stages, capturing the discrepancy from both perspectives: from the source to the target, and vice versa.

In the first stage, for each sampled point on the source mesh, the nearest point on the target mesh is retrieved using the `search_knn_vector_3d` method provided by the `Open3D` library. The unsigned Euclidean distance is then computed. To introduce directional information, the sign of the distance is determined by

evaluating the dot product between the distance vector (from the source point to the target point) and the surface normal of the source point. If this dot product is close to zero (indicating near-perpendicularity between the vector and the normal), the normal of the nearest target point is used instead. This fallback strategy improves robustness in regions of ambiguous local geometry. The final signed distance is negative when the point lies "inside" the target surface and positive otherwise. The implementation is described in the Listing 5.1

```
dist_vector = nearest_point - point
dot_product = np.dot(dist_vector, normal)

if abs(dot_product) <= 0.15:
    dot_product = np.dot(dist_vector, nearest_point_normal)

signed_distance = dist if dot_product >= 0 else -dist
```

Listing 5.1 Signed distance computation

In the second stage, the same procedure is repeated in reverse. For each point in the target point cloud, the five nearest neighbors on the source mesh are identified. For each of these neighbors, the signed distance is computed using the same method described above. The newly computed distances are averaged with the previously stored ones (of the source point cloud) from the first pass. This bidirectional approach, inspired by the Chamfer distance computation, helps ensure that both protrusions and indentations are captured effectively, including regions that might otherwise be missed in a one-directional comparison.

Colorize by Distance

After computing the signed distances between the source and target meshes, each point in the source point cloud is associated with the distance value. To enable a meaningful visual representation of these values, a colorization process is applied based on threshold parameters provided via command-line input.

Specifically, two sets of three threshold values are defined: one for positive distances and one for negative distances. These thresholds serve as breakpoints for mapping the distance values to corresponding color intervals. Positive distances are mapped to a warm color palette (e.g., yellow to red). On the other hand, negative distances are mapped to a cold color palette (e.g., light blue to dark blue).

Each distance value is assigned a color by linearly interpolating between the corresponding breakpoints within the appropriate palette. This approach ensures a smooth and continuous gradient that highlights the extent of deviation across the surface. Distances close to zero, which indicate good alignment between the two meshes, are mapped to white.

The computed colors are stored within the `SampledMesh` class, maintaining the association between each sampled point and its respective color value. These colored points are later used to generate the final texture.

Point Color to Texture Mapping

The final operation performed by the back-end involves the generation of a sparse texture that encodes the per-point color information in the two-dimensional UV space of the mesh. This mapping enables the transfer of 3D distance information onto a texture that can be rendered efficiently in the visualization stage.

For each sampled point, the triangle (face) on which the point lies is retrieved using the face indices stored during the sampling process. The barycentric coordinates of the point with respect to its corresponding triangle are then computed. These coordinates are used to interpolate the UV coordinates of the triangle's vertices, thereby obtaining the precise UV position of the point.

Once the UV position is determined, it is mapped to discrete texture coordinates (in pixel space), and the corresponding pixel in the texture image is colored using the RGB value previously assigned to the point during the colorization phase.

In addition to this per-point mapping, each vertex of the triangle is also assigned the color of the first sampled point associated with the face. This fallback color is stored to be used later during the rendering stage, particularly in cases where the interpolated texture may lack coverage or where shading artifacts need to be mitigated.

5.4.3 Front-End Rendering Module

The front-end component of the system is implemented in Unity, with particular attention paid to the constraints imposed by WebGL builds, especially for custom shaders.

The primary responsibility of the front-end module is to render the processed 3D models and to reconstruct a continuous heatmap texture starting from the sparse, point-based texture generated by the back-end. This reconstructed texture is then mapped onto the surface of the 3D model using the UV coordinates computed during preprocessing.

In addition to visualization, the front-end manages the dynamic loading of assets from the server, including both geometry and associated texture maps. Lighting and

shading are configured to enhance the readability of the heatmap while preserving the geometric detail of the object. Where appropriate, transparency and overlay techniques are used to allow the visual comparison between reconstructed and reference models.

Design Constraints

In order to implement the texture filling operation efficiently, a custom shader was developed to use GPU-based computation, which is inherently optimized for operations on textures and rendered surfaces. However, utilizing such computations introduces several constraints, primarily due to the use of WebGL builds and the requirement for interactive rendering within a browser environment.

An initial evaluation of different implementation strategies was conducted to balance performance and visual accuracy. The first approach involved a fragment shader that receives as input a vector of 3D points (coordinates x, y, z) and an associated vector of color values. For each fragment, the shader transforms the UV coordinates into object space and identifies the closest point to determine the appropriate color. While this method is conceptually straightforward, it proved to be poorly scalable: as the number of input points increases, the computational cost for nearest neighbor search grows significantly, resulting in severe performance degradation. As shown in the Figure 5.4, with more than 20,000 points, the project is less and less interactive.

To address this limitation, a second strategy was developed in which the fragment shader operates directly in UV space. Instead of comparing with all input points, each fragment searches for the nearest colored pixel in a progressively expanding neighborhood. This approach is computationally efficient and scales well, as a higher point density results in a shorter search. However, this method suffers from visual artifacts due to UV space topology: adjacent UV regions do not necessarily correspond to neighboring regions in 3D space. Consequently, fragments near triangle edges may sample colors from unrelated, nearby faces. An example of this issue is shown in the Figure 5.5

To mitigate this artifact, a constrained search strategy was introduced, whereby the nearest-colored pixel is searched only within the face to which the fragment belongs. This requires identifying the triangle in which the fragment lies. To achieve this, a geometry shader was employed to output the three vertex coordinates of each triangle to a data structure accessible in the fragment shader. These coordinates are then used to constrain the filling logic. Unfortunately, this solution is incompatible with WebGL, which does not support geometry shaders.

The final implementation maintains the principle of nearest-colored pixel search, but without using the geometry shader. Specifically, Unity's `Graphics.Blit()` function is employed to simulate a compute shader using a fragment shader. This

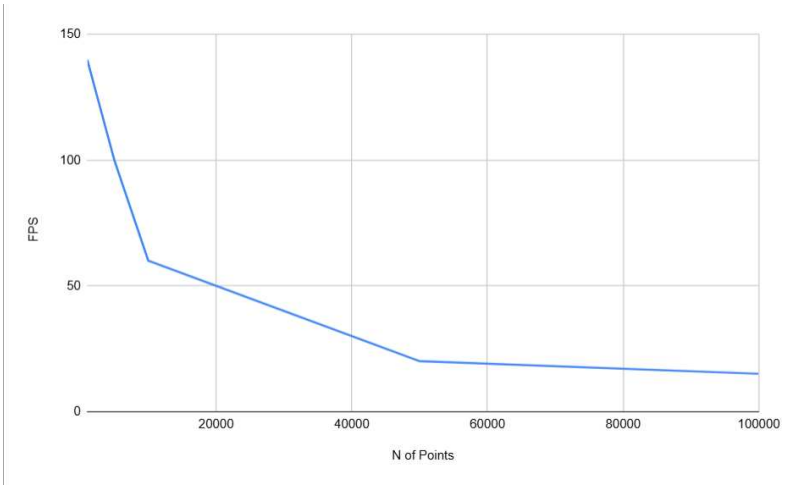
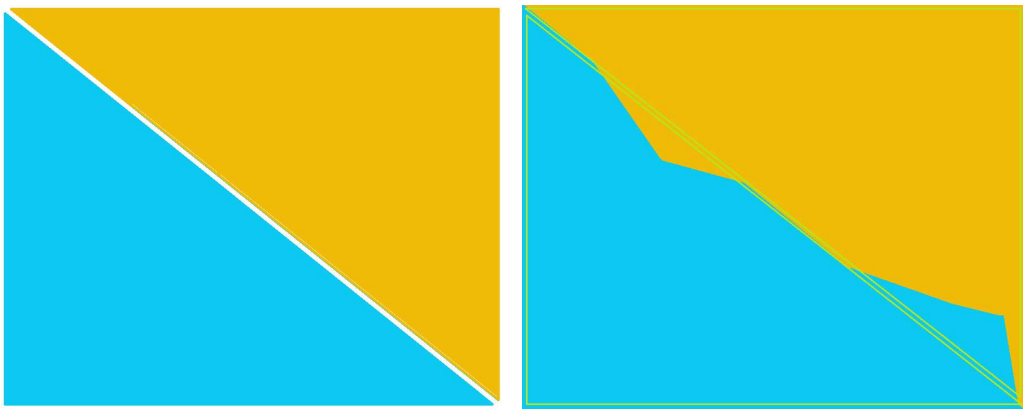


Figure 5.4: Frames per second with respect of number of points used as input of the shader



(a) (b)

Figure 5.5: Example of color spilling on the edge of the faces. (a) On the left the expected result and (b) on the right the actual outcome. The green triangles on the right indicate the face edge

function executes a full-screen render pass where each pixel of the output texture is computed independently using custom logic defined in the shader. Through this mechanism, GPGPU-style parallelism is achieved within the limitations of the WebGL pipeline, enabling efficient texture reconstruction with high visual fidelity.

Implementation Details

The front-end rendering module is structured around a main controlling script and two dedicated shaders that respectively perform the computation and visualization of the heatmap.

The central component is the `ModelDiffVisualizer` script, which orchestrates the loading of resources and initialization of materials. This script is responsible for managing interactions between the different modules and handling data exchange between CPU and GPU memory.

The texture filling logic is implemented in the `HeatmapCompute.shader`, a fragment shader that simulates GPGPU-style computations via a full-screen pass. This shader is executed through Unity's `Graphics.Blit()` function.

Finally, the `HeatmapVisualize.shader` is responsible for rendering the final result on the 3D mesh by applying the filled texture and mapping it onto the mesh surface according to the UV layout. This shader also includes the shading logic to enhance visual readability of the reconstructed texture.

In the following sections, the specific implementation strategies for each of these components are described in detail.

HeatmapCompute.shader The `HeatmapCompute.shader` is a GPU shader employed within Unity's `Graphics.Blit()` function, effectively acting as a compute operation on a per-pixel basis over a render texture. The shader consists of a basic vertex shader and a fragment shader. The vertex shader performs a standard transformation of vertices using the built-in `UnityObjectToClipPos`, also assigning the UV coordinates to the output struct. Since no 3D geometry is actually rendered in this context, the vertex shader is effectively preparing a full-screen quad for fragment processing. The vertex shader function is illustrated in the Listing 5.2

```
1 v2f vert(appdata v)
2 {
3     v2f o;
4     o.pos = UnityObjectToClipPos(v.vertex);
5     o.uv = v.uv;
6     return o;
7 }
```

Listing 5.2 Vertex shader of HeatmapCompute.shader

The core logic is implemented in the fragment shader, where the computation

proceeds pixel by pixel in UV space. The shader receives multiple inputs:

- A *point texture* containing the colors assigned to sampled 3D points;
- A *position mask texture* where white pixels correspond to valid colored points in the point texture;
- A *UV texture* encoding the UV coordinates of triangle vertices in a packed 2D format;
- A *maximum radius* defining the search extent in pixels.

Due to WebGL limitations, vertex UVs could not be passed directly as arrays or buffers. Instead, a workaround was adopted, converting the UV coordinate list into a floating-point 2D texture, where each texel stores a `float2` corresponding to a UV coordinate. The C# code used to generate this UV texture is reported in Listing 5.3.

```
1  int texWidth = Mathf.NextPowerOfTwo(
2          Mathf.CeilToInt(Mathf.Sqrt(uvCount))
3      );
4  int texHeight = Mathf.CeilToInt((float)uvCount / texWidth);
5  Texture2D uvTex = new Texture2D(texWidth, texHeight,
       TextureFormat.RGFloat, false);
6  for (int i = 0; i < uvCount; i++) {
7      int x = i % texWidth;
8      int y = i / texWidth;
9      uvTex.SetPixel(x, y, new Color(uv[i].x, uv[i].y, 0));
10 }
11 uvTex.Apply();
```

Listing 5.3 Generation of the UV coordinate texture in Unity

The UV values are retrieved in the shader with a dedicated function, shown in Listing 5.4, which reconstructs the UV coordinates from the packed texture using the index and the known texture resolution.

The first computational step in the fragment shader consists of identifying the triangle in UV space to which the current fragment belongs. This is achieved by iterating through sets of three UV vertices at a time and checking whether the

```
float2 getUV(uint index)
{
    uint x = index % (int)_UVTexSize.x;
    uint y = index / (int)_UVTexSize.x;
    float2 uv = (float2(x, y) + 0.5) / _UVTexSize;
    return tex2D(_UVTex, uv).rg;
}
```

Listing 5.4 Reconstructing UV coordinates in the shader

fragment's UV lies inside the triangle defined by them. Upon finding a match, the three UV vertices of the corresponding triangle are cached.

Following the triangle detection, the shader performs a spiral search in UV space, centered on the fragment's coordinates. The search progressively expands in concentric square layers, checking each neighbor pixel until a colored point is found. Each candidate pixel is validated against three conditions: it must be inside the UV bounds, it must be marked as colored in the position texture, and it must lie inside the triangle identified in the previous step. The checking condition is illustrated in the Listing 5.5.

```
if (uvInBound(searchUV) &&
    is_colored(get_color(_Positions, searchUV)) &&
    pointInTriangle(searchUV, uv0, uv1, uv2))
{
    return get_color(_PointsTex, searchUV);
}
```

Listing 5.5 Check executed for the closest colored point inside the triangle

The colored pixel search is explained in the Figure 5.6.

Once a matching colored point is found, its color is written to the output texture. If no point is found within the maximum radius, the output pixel is assigned a transparent value. This strategy guarantees interactive performance and visual coherence while respecting the rendering constraints imposed by WebGL builds.

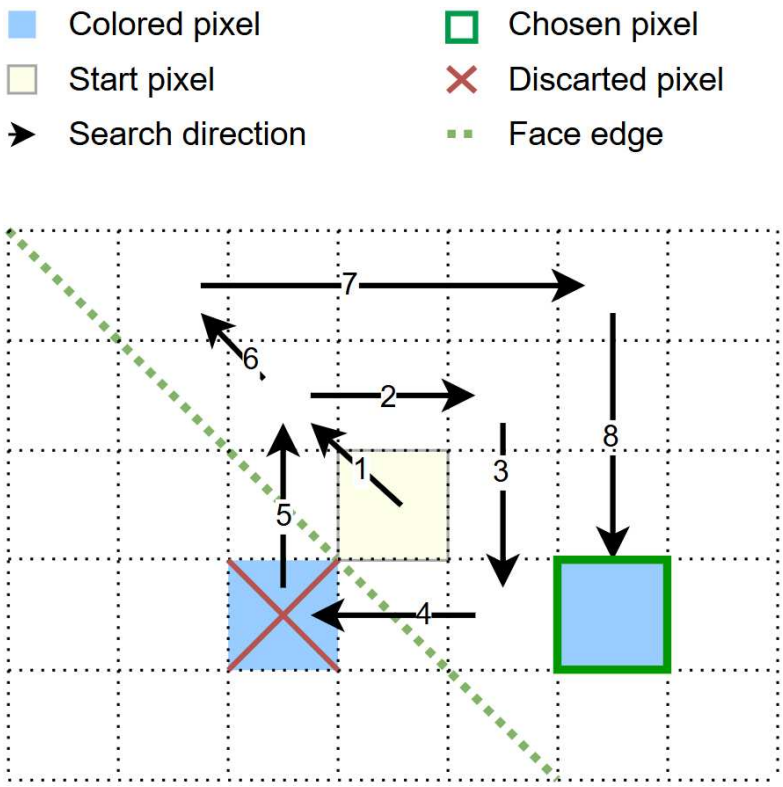


Figure 5.6: Colored pixel search workflow

HeatmapVisualize.shader The `HeatmapVisualize.shader` is responsible for correctly displaying the heatmap texture over the corresponding 3D model. It is implemented as a custom multi-pass shader, structured into three distinct passes: a depth-only pass, the main color rendering pass, and an outline post process pass.

- **Pass 1 – Depth Pass**: This pass writes only to the depth buffer, discarding any color output. Its purpose is to ensure that the model is properly occluded and can be rendered with transparency without showing internal faces or artifacts from back-facing geometry.
- **Pass 2 – Main Rendering Pass**: This is the core of the heatmap visualization logic. It samples the heatmap texture using the fragment's UV coordinates. If the sampled color is transparent, a fallback color is applied by interpolating the per-vertex fallback colors, which are passed from the vertex shader. The result is then modulated with a diffuse lighting model to enhance the 3D perception of the surface. The lighting is calculated using the dot product between the surface normal and the light direction, corresponding to Lambertian diffuse shading [51]. The shaded color is then re-modulated with

arbitrary weights to achieve a clearer visualization, as shown in the Listing 5.6.

```
half4 baseColor = color;
half3 normal = normalize(i.normal);

// diffuse lighting
half3 lightDir = normalize(_WorldSpaceLightPos0.xyz);
half  diff = max(0, dot(normal, lightDir));
half  inverse_diff = max(0, dot(normal, -lightDir));

half3 lighting = baseColor.rgb * 0.02 +
                 0.98 * (baseColor.rgb * _WorldColor.rgb
                 + diff * baseColor.rgb * 0.9 -
                  inverse_diff * baseColor.rgb * 0.1);

return half4(lighting, baseColor.a);
```

Listing 5.6 Diffuse lighting applied to heatmap color

Although this shading model is neither physically-based nor photorealistic, it was designed to improve depth perception and ensure color uniformity across the model. This avoids overly bright or dark regions that could impair the readability of the heatmap.

- **Pass 3 – Outline Pass**: The final rendering pass introduces a stylized outline around the model to improve edge visibility and silhouette clarity. To achieve this, the mesh is rendered a second time in black, slightly enlarged in clip space. This is done by displacing each vertex outward along its normal before projection. As a result, the outline appears as a black border beneath the primary model surface.

The end result is displayed in the Figure 5.7

ModelDiffVisualizer The `ModelDiffVisualizer` script handles the loading and preparation of 3D models and associated data required for the correct rendering of the heatmap.

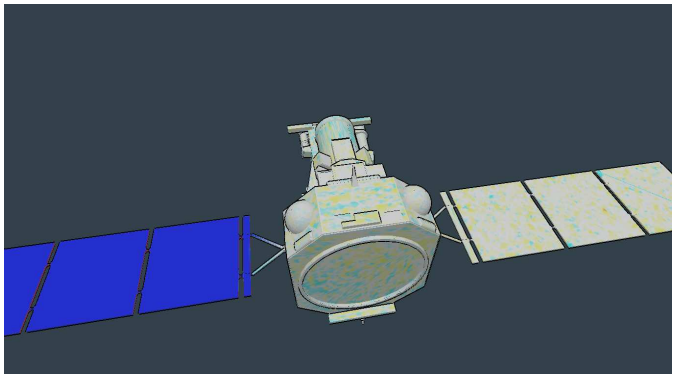


Figure 5.7: Example of the final result of the heatmap shader. The heatmap shows that the scanned model is missing a piece

Initially, the `source` and `target` models are loaded based on URLs provided as input parameters. This operation is performed using a customized version of Unity's `OBJImporter` asset, which was extended to also import per-vertex color information, ensuring consistency with the preprocessing pipeline.

After the models are loaded, the script downloads the auxiliary textures produced during the back-end processing stage: the `points`, `positions`, and `UVs` textures. The UV texture is constructed programmatically, as described in previous sections, by encoding the UV coordinates of the face vertices into a texture.

Once all data is prepared, a call to `Graphics.Blit()` is issued using the material configured with the `HeatmapCompute.shader`. This operation simulates a GPGPU computation, producing the final heatmap texture by executing the fragment shader logic on a per-pixel basis across the render target.

Finally, the generated heatmap texture is applied to the `source` model using the `HeatmapVisualize` material, enabling the colored visualization of differences. The `target` model is assigned a semi-transparent material, allowing its shape to remain perceptible while not interfering with the readability of the source model's heatmap.

Chapter 6

Demo

To validate the developed tools and to provide a comprehensive example of their integration, a demonstration scenario has been implemented. This demo serves both as a practical usage case and as a guideline for the application of the proposed framework in aerospace-related projects.

The primary goal of the demonstration is to illustrate the complete pipeline, from data acquisition to real-time visualization, using the components developed in this thesis. The scenario is based on a simulated operation in which a LiDAR sensor rotates around a satellite and acquires 3D scans of its surface. The operation is intentionally fictitious yet representative of potential use cases in space missions involving close-range inspection or servicing tasks.

The architecture of the system follows a client-server paradigm. The server component is responsible for handling telemetry and reconstruction. It transmits live telemetry data to the client, which visualizes the scanning operation in real time. Once the acquisition phase is completed, the server performs the 3D reconstruction of the target and sends the resulting mesh to the client. On the client side, the reconstructed model is rendered using the heatmap visualization.

The final demo page is shown in the Figure 6.1

6.1 Page Layout

The demonstration interface is organized into three main sections.

The upper portion of the page shows the reference model used to make the comparison with the scanned one, selected in the middle section. The middle portion of the page focuses on the live simulation. The user can select a specific model to scan,1 and then it displays a real-time visualization of the scanning process and includes a telemetry table. This table presents a tabular view of the data transmitted during the simulated mission, such as the current sensor position,

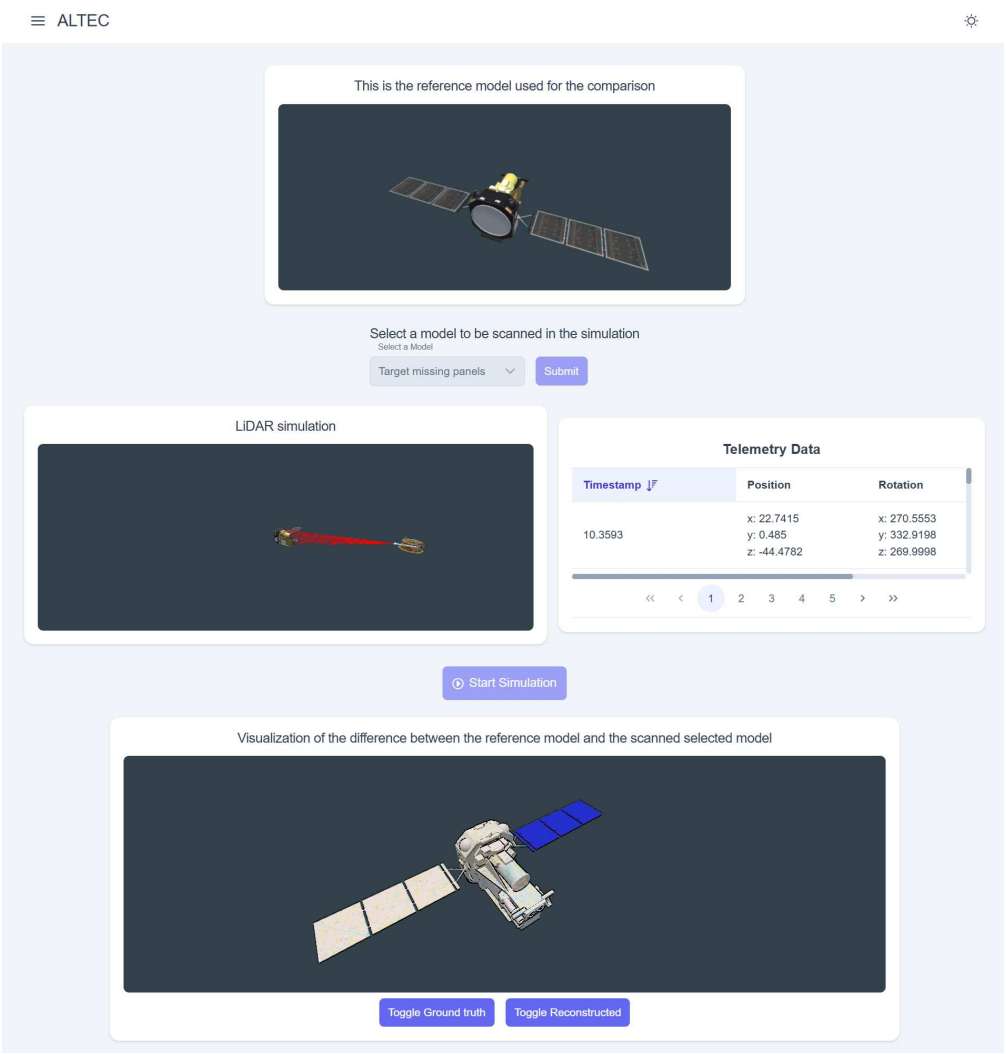


Figure 6.1: Demo page

rotation, or time stamps. The simulation provides users with a clear understanding of the acquisition workflow and the evolution of the scanning process over time.

The lower portion of the interface hosts an embedded Unity instance responsible for rendering the reconstructed model using the developed heatmap visualization technique. This area allows users to inspect the results of the 3D reconstruction and to interpret deviations or inconsistencies with respect to the reference geometry.

At the beginning of the simulation, users are prompted to select a model that will serve as the target for the scanning process. The reference object will be always the same; the selected model, however, represents the scanned geometry and is crucial for evaluating the effectiveness of the heatmap visualization. By choosing

different target models, users can observe how geometric discrepancies influence the color patterns of the heatmap.

6.2 Back-end Implementation

The back-end component of the system is implemented using the NestJS framework. Its responsibilities are limited to providing access to 3D models, simulating live telemetry, and executing scripts for mesh reconstruction and preprocessing for heatmap visualization.

The back-end exposes the following endpoints:

- `/api/meshes`: Returns the URLs of the 3D meshes used as satellite targets for the simulation. These files are publicly accessible and can be downloaded by the client.
- `/api/reconstruct`: Triggers the reconstruction of the scanned mesh. It assumes the availability of the raw point cloud data on the server and returns the URL to the resulting reconstructed model.
- `/api/generate-difference`: Performs preprocessing required for heatmap visualization. It assumes the reconstructed model is already present on the server and returns the paths to the generated texture and model.
- `WebSocket`: a WebSocket is provided for streaming telemetry data to the client in real time.

Further explaining the WebSocket functionalities, upon connection, the server waits for a `start-stream` message. Once received, it reads a set of mock telemetry records and sends them to the client, respecting the original time intervals between consecutive data entries, as shown in the Listing 6.1

Each telemetry packet includes information such as the timestamp, the position and orientation of the satellite carrying the LiDAR sensor, and the rotation of the joints on which the sensor is mounted. When the data stream is complete, an `end` message is emitted.

The `/api/reconstruct` endpoint executes a script that performs the surface reconstruction using Screened Poisson Surface Reconstruction (SPSR), which was identified as the best-performing method during algorithm benchmarking. The script assumes that the input point cloud is available in a known directory. The output mesh is saved in a dedicated directory that changes based on the selected target model. If the output already exists, the reconstruction is skipped. Each model has a unique identifier passed to the endpoint to correctly select the corresponding paths.

```
1 if (prevTimestamp) {
2     await this.timeout((row.Timestamp - prevTimestamp) *
    1000);
3 }
4 prevTimestamp = row.Timestamp;
5 this.server.emit('telemetryData', this.telemetryToDto(row));
```

Listing 6.1 Telemetry data emission with timing control

Similarly, the `/api/generate-difference` endpoint runs a script that preprocesses geometric differences and generates the necessary textures for heatmap rendering. This script is based on the method described in chapter 5. Output files are cached and reused if they already exist.

Internally, the back-end integrates the reconstruction and preprocessing scripts directly. The reconstruction script is a Python wrapper around a modified C++ implementation of SPSR. The wrapper validates input files, converts formats as needed (from PLY to OBJ and vice versa), and launches the reconstruction with optimized parameters identified during previous testing.

The back-end is designed to be containerized using Docker, allowing ALTEC to deploy the complete system in a reproducible environment. A `docker-compose` setup is provided, structured in two stages:

- The first stage builds an image (`build-externals`) that installs external dependencies and compiles the SPSR C++ project.
- The second stage uses this image as a base, adds the NestJS project, and launches the back-end service.

An excerpt of `docker-compose.yml` configuration is reported in the Listing 6.2

Two Dockerfiles are used: one for building the external tools, and one for configuring the full NestJS service. The build-externals Dockerfile installs Python dependencies for mesh comparison and compiles the SPSR executable, as explained in the Listing 6.3

This setup ensures that all necessary libraries and scripts are available within the container, enabling a smooth deployment of the back-end services and reconstruction pipelines.

```
services:
  nestjs:
    (...)
    depends_on:
      - externals-scripts
    volumes:
      (...)
      - externals-volume:/app/PoissonRecon
  externals-scripts:
    image: build-externals
    volumes:
      - externals-volume:/app/Poisson
      Recon
volumes:
  externals-volume:
```

Listing 6.2 Docker Compose configuration

6.3 Front-end Implementation

The front-end web application was developed using the Vue.js framework. The project is composed of a main view that orchestrates the overall layout and logic, supported by a set of auxiliary components. Two of these components encapsulate the Unity WebGL instances used respectively for the live simulation (`LoadAndScanModelUnity.vue`) and for the mesh difference visualization (`ModelDiffUnity.vue`). These components make use of the Unity-Vue integration library developed as part of this thesis project.

Additionally, the front-end includes a network service class for managing API interactions and a composable module, `useTelemetry.ts`, for handling real-time telemetry data.

6.3.1 Telemetry Management with `useTelemetry.ts`

The `useTelemetry` composable is designed to manage the connection with the back-end WebSocket and handle the telemetry stream. It is built to be reusable by both the Unity simulation component and the telemetry table component. Upon initialization, the composable establishes the WebSocket connection and registers

a listener for telemetry data messages. When new data is received, a callback function is invoked, and the data is appended to a local telemetry history buffer.

To optimize performance and reduce excessive reactivity updates, the history is updated periodically using Vue's throttling mechanism. As shown in the Listing 6.4, buffered data is committed to the reactive array every 300 ms.

```
1  FROM node:22 AS base
2  (...)
3  FROM python:3.11-slim AS python-deps
4  (...)
5  COPY mesh-difference/Pipfile .
6  RUN pipenv install
7  FROM base AS mesh-reconstruction
8  COPY --from=python-deps /.venv /.venv
9  ENV PATH="/.venv/bin:$PATH"
10 RUN ln -sf $(which python3) /usr/local/bin/python
11 FROM mesh-reconstruction
12 (...)
13 RUN make poissonrecon -j 3
```

Listing 6.3 Multi-stage Dockerfile for installing external libraries

```
1 const handleNewData = Utils.throttle(() => {
2   telemetryHistory.value.push(...historyBuffer);
3   historyBuffer = [];
4 }, 300);
```

Listing 6.4 Throttled data handler for telemetry history updates

6.3.2 Vue component `LoadAndScanModelUnity`

This component manages the WebGL instance for the real-time simulation. Initially, it constructs a configuration object with the URL of the Unity WebGL build to be

passed to the Unity component. During initialization, it displays a loading spinner showing the current loading progress of the Unity instance.

Externally, the component exposes:

- an event indicating when the model has been successfully loaded,
- a function to trigger Unity's model loading process,
- and a function to start the simulation.

6.3.3 Vue component `ModelDiffUnity`

Similarly, this component configures the Unity WebGL instance and manages its loading status using a progress indicator. Once loaded, it exposes several methods to the parent view:

- a method to invoke Unity's function for visualizing differences between the source and target meshes,
- methods to toggle the visibility of the source and target models, facilitating a clearer interpretation of the heatmap.

6.3.4 Main View and User Interaction

The main view coordinates the entire interface. Displays the reference model, presents a dropdown selector to choose the target satellite model, and buttons to initiate both the simulation and the reconstruction process. It also includes controls for toggling the visibility of the source and target meshes in the second Unity instance. This layout allows the user to simulate the full pipeline, from selecting a model and visualizing the simulated LiDAR scan to comparing the reconstructed mesh using heatmap visualization.

6.3.5 Unity Project Implementation

The Unity project developed for the demo project supports both real-time simulation of the LiDAR scanning operation and visualization of differences between 3D models. In order to maintain a unified codebase, both functionalities were implemented within a single Unity project. The two use cases were structured as separate scenes and compiled into distinct WebGL builds, each dedicated to a specific functionality.

Simulation Scene

This scene manages the simulation of the scanning operation. It displays the target satellite model and updates the sensor's position and orientation in real time according to telemetry data received from the Vue frontend.

To support dynamic model loading, the simulation scene includes a script that loads the 3D model at runtime based on a URL received from the frontend. This operation is performed using the `GLTFast` library, as shown in the Listing 6.5

```
1 var gltf = loader.AddComponent<GltfAsset>();
2 var result = await gltf.Load(url);
```

Listing 6.5 Loading of a GLTF model at runtime

The main control script, `GameManager`, exposes the method `UpdateSimulation`, which is invoked by the Vue application. This function accepts a telemetry object serialized in JSON due to the restriction that Unity WebGL can only receive primitive types from JavaScript.

Upon receiving a new telemetry object, Unity stores it without immediately updating the simulation state. Instead, the system performs a temporal interpolation between consecutive telemetry updates. Specifically, the time difference between the current and previous telemetry timestamps is calculated, and the sensor's position and rotation are smoothly interpolated over this interval. This ensures a visually smooth transition between updates and prepares the simulation for the next interpolation cycle upon receiving a new telemetry sample.

To enhance the visual representation of the LiDAR scanning operation, a collection of rays is displayed emanating from the sensor module, simulating the laser beams of a LiDAR device, as illustrated in the Figure 6.2.

At the end of the simulation, the point cloud resulting from the scan is displayed as shown in the Figure 6.3. The point cloud is visualized using a set of spheres whose brightness varies depending on the orientation of the surface normal relative to the camera view: points whose normals face the camera appear brighter, while those facing away appear darker. This approach provides a visual cue for the orientation of the points.

GPU instancing, via the Unity `DrawMeshInstanced()` function, is used to render the point cloud efficiently. A custom shader was implemented that takes as input a vector of normals and a set of 4×4 transformation matrices representing the spatial position and orientation of each point.

The vertex shader computes the dot product between each point's normal and the camera's view direction. Each point is then rendered by transforming a base

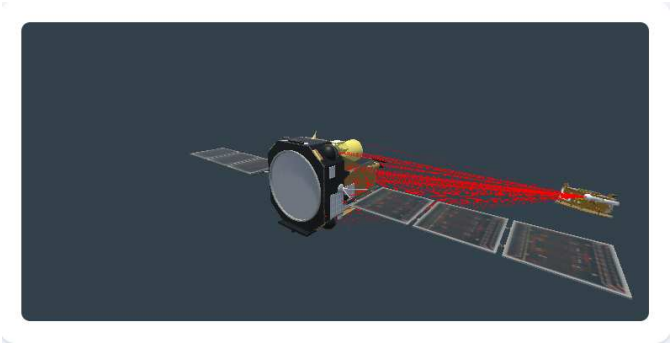


Figure 6.2: Demo simulation. A group of red rays is displayed to indicate the scanning procedure

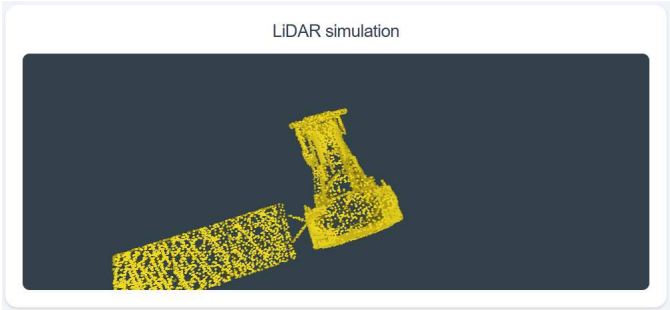


Figure 6.3: Point cloud representing the scanned object

sphere mesh using the corresponding matrix. Additionally, the scale of the spheres is dynamically adjusted according to the camera's distance, ensuring that points do not appear disproportionately small or large when zooming. Finally, the color intensity is derived from the dot product result, linearly mapped to a brightness range between 0.1 and 1.

Model Difference Scene

The second scene focuses on the visualization of structural differences between the reference and reconstructed models. This component builds upon the Unity-based tool described in chapter 5. At runtime, both the source and target 3D models are loaded, along with the corresponding texture maps. These assets are retrieved using URLs provided by the Vue frontend.

Once loaded, the scene exposes a set of functions that allow the user to toggle the visibility of the source and target models.

Chapter 7

Conclusions

Two tools and a library have been developed as part of this work, conducted at ALTEC, to support the analysis, reconstruction, and visualization of 3D data in simulated aerospace scenarios.

First, a custom TypeScript library was implemented to embed Unity WebGL builds into a Vue.js web application. The library abstracts the complexities of managing the Unity instance lifecycle and establishes a communication layer that enables bidirectional data exchange between the front-end and the simulation environment. This design facilitates the integration of interactive Unity content into a web page. Tests were also conducted to evaluate the performance of the library.

Second, a test and benchmarking framework was designed to evaluate and compare remeshing algorithms under controlled and reproducible conditions. The framework extends an existing open-source benchmark and includes a synthetic data generation pipeline based on a virtual LiDAR simulator. This framework enables systematic parameter exploration and metric-based evaluation of the quality of the reconstruction.

Third, a method for visualizing geometric discrepancies between 3D models was developed to support the detection of potential structural anomalies. The approach is based on computing deviations between a scanned mesh and its CAD counterpart and encoding the differences through a heatmap representation directly rendered on the 3D surface. This technique provides immediate visual feedback and facilitates the identification of missing parts, deformations, or unexpected artifacts in the scanned geometry.

Finally, a demonstration scenario was created to integrate and validate the developed tools. The demo simulates a LiDAR scanning operation, where telemetry is streamed from the back-end to the client application, which then visualizes the process in real time. After the reconstruction phase, the resulting mesh is displayed alongside its reference model using the proposed heatmap visualization, offering a

complete end-to-end representation of the pipeline.

These contributions provide ALTEC with a foundation for future research and development in the field of 3D reconstruction and model comparison for aerospace applications and simulations. The tools developed in this work facilitate the integration of simulation, evaluation, and visualization within a web framework. Modular architecture and dockerization promote extensibility and ease of integration into broader mission support systems.

7.1 Future Work

Several options are available to extend and improve the tools developed in this work, both in terms of functionality and performance. This section outlines the most relevant enhancements identified during the development process.

Unity Integration Library. One promising extension of the TypeScript library for Unity integration is the implementation of a reload mechanism for Unity instances.

This would allow developers to reset or reload a simulation without requiring a full page refresh.

LiDAR Simulator and Remeshing Framework. Future work on the simulation and benchmarking framework could explore the adoption of more advanced reconstruction techniques. For instance, neural network-based remeshing algorithms trained on custom datasets may provide improved generalization and robustness, especially when dealing with noisy or partial data.

Furthermore, the LiDAR simulator can be refined by evaluating the influence of different acquisition parameters, such as sensor path inclination, resolution, number of revolutions, frame rate, and final point cloud density.

The noise model could also be adjusted to more closely approximate the behavior of real-world depth sensors, incorporating factors such as surface reflectivity and incidence angles.

Visualization Tools. At present, modifying breakpoints requires recomputation of the color mapping server-side; moving this logic to the GPU via shader programming would make the process fully interactive and more responsive.

The current visualization approach could be extended by delegating the generation of a continuous intermediate map to the back-end. This would enable the front-end to render the heatmap with interactive controls selecting color palettes and adjusting value breakpoints without reprocessing.

Additional improvements include allowing users to select from different distance metrics (e.g., point-to-point, point-to-plane, etc.), as well as offering customizable color schemes to better adapt the visualization to specific analytical needs or visual preferences.

Glossary

LiDAR
: Laser Detection and Ranging

3D
: Three dimensional

CAD
: Computer-aided design

WebGL
: Web-based Graphics Library

API
: Application programming interface

UI
: User Interface

CPU
: Central Processing Unit

FOV
: Field Of View

GPU
: Graphics Processing Unit

VR
: Virtual Reality

AR

Augmented Reality

XR

Extended Reality

MR

Mixed Reality

AV

Augmented Virtuality

WBS

Web-based simulation

SIFT

Scale-Invariant Feature Transform

Bibliography

[1] *ALTEC Website | project*. en-US. URL: https://www.altecspace.it/ (visited on 05/12/2025) (cit. on p. 2).

[2] Grigore C. Burdea and Philippe Coiffet. *Virtual Reality Technology*. en. John Wiley & Sons, June 2003. ISBN: 978-0-471-36089-6 (cit. on p. 6).

[3] Paul Milgram, Haruo Takemura, Akira Utsumi, and Fumio Kishino. «Augmented reality: a class of displays on the reality-virtuality continuum». In: *Telemanipulator and Telepresence Technologies*. Vol. 2351. SPIE, Dec. 1995, pp. 282–292. DOI: 10.1117/12.197321. URL: https://www.spiedigitallibrary.org/conference-proceedings-of-spie/2351/0000/Augmented-reality--a-class-of-displays-on-the-reality/10.1117/12.197321.full (cit. on p. 7).

[4] Yevgeniy Zhivotovskiy, Emily Stratton, Shean Phelps, and Josef Schmid. *The Reality of Space: Exploring Virtual, Augmented, and Mixed Realities and the Future of Spaceflight*. NTRS Author Affiliations: Arkansas College of Osteopathic Medicine, The University of Texas Medical Branch at Galveston, Johnson Space Center NTRS Document ID: 20240005474 NTRS Research Center: Johnson Space Center (JSC). URL: https://ntrs.nasa.gov/citations/20240005474 (cit. on pp. 8–10).

[5] *Better than Reality: NASA Scientists Tap Virtual Reality to Make a Scientific Discovery - NASA*. en-US. Section: Goddard Technology. Jan. 2020. URL: https://www.nasa.gov/technology/goddard-tech/virtual-reality-enables-discovery/ (visited on 06/08/2025) (cit. on p. 8).

[6] Angelica D. Garcia, Jonathan Schlueter, and Eddie Paddock. «Training Astronauts using Hardware-in-the-Loop Simulations and Virtual Reality». en. In: *AIAA Scitech 2020 Forum*. Orlando, FL: American Institute of Aeronautics and Astronautics, Jan. 2020. ISBN: 978-1-62410-595-1. DOI: 10.2514/6.2020-0167. URL: https://arc.aiaa.org/doi/10.2514/6.2020-0167 (visited on 06/08/2025) (cit. on p. 9).

[7] *Ready, set, go for COVID-conscious astronaut training*. en. URL: https://www.esa.int/About_Us/EAC/Ready_set_go_for_COVID-conscious_astronaut_training (visited on 06/08/2025) (cit. on p. 10).

[8] Tanya Andrews, Brittani Searcy, and Brianna Wallace. «Using Virtual Reality and Motion Capture as Tools for Human Factors Engineering at NASA Marshall Space Flight Center». en. In: *Advances in Artificial Intelligence, Software and Systems Engineering*. Ed. by Tareq Ahram. Vol. 965. Series Title: Advances in Intelligent Systems and Computing. Cham: Springer International Publishing, 2020, pp. 399–408. DOI: 10.1007/978-3-030-20454-9_41. URL: http://link.springer.com/10.1007/978-3-030-20454-9_41 (visited on 06/08/2025) (cit. on p. 10).

[9] *Deep Space Habitation Overview*. en-US. Apr. 2016. URL: https://www.nasa.gov/humans-in-space/deep-space-habitation-overview/ (cit. on p. 10).

[10] *Kennedy Space Center - NASA*. en-US. URL: https://www.nasa.gov/kennedy/ (cit. on p. 10).

[11] *What Is a Digital Twin?* en. Aug. 2021. URL: https://www.ibm.com/think/topics/what-is-a-digital-twin (cit. on p. 10).

[12] Lucio Pinello, Lorenzo Brancato, Marco Giglio, Francesco Cadini, and Giuseppe Francesco De Luca. «Enhancing Planetary Exploration through Digital Twins: A Tool for Virtual Prototyping and HUMS Design». en. In: *Aerospace* 11.1 (Jan. 2024), p. 73. ISSN: 2226-4310. DOI: 10.3390/aerospace11010073. URL: https://www.mdpi.com/2226-4310/11/1/73 (visited on 06/08/2025) (cit. on pp. 10, 11).

[13] *Interplanetary Twins*. en. URL: https://www.sfmagazine.com/technotes/2021/march/interplanetary-twins/ (cit. on p. 11).

[14] *NASA Readies Perseverance Mars Rover's Earthly Twin - NASA*. en-US. Sept. 2020. URL: https://www.nasa.gov/centers-and-facilities/jpl/nasa-readies-perseverance-mars-rovers-earthly-twin/ (cit. on p. 11).

[15] James Byrne, Cathal Heavey, and P. J. Byrne. «A review of Web-based simulation and supporting tools». In: *Simulation Modelling Practice and Theory* 18.3 (Mar. 2010), pp. 253–276. ISSN: 1569-190X. DOI: 10.1016/j.simpat.2009.09.013 (cit. on p. 11).

[16] Zhongming Zhang, Jian Xu, Xun Shen, Hailing Zhao, and Yaohui Niu. «WebGL-based virtual reality technology construction and optimization». In: *International Conference on Optics, Electronics, and Communication Engineering (OECE 2024)*. Vol. 13395. SPIE, Nov. 2024, pp. 393–400. DOI: 10.1117/12.3048388. URL: https://www.spiedigitallibrary.org/con

ference-proceedings-of-spie/13395/133951L/WebGL-based-virtual-reality-technology-construction-and-optimization/10.1117/12.3048388.full (cit. on p. 11).

[17] *Enterprise JavaBeans Technology*. URL: https://www.oracle.com/java/technologies/enterprise-javabeans-technology.html (cit. on p. 11).

[18] *Applets*. URL: https://www.oracle.com/java/technologies/applets.html (cit. on p. 11).

[19] *GitHub - martimy/JSim: JSim is a Java-based discrete event simulator of an M/M/s queue system*. URL: https://github.com/martimy/JSim (cit. on p. 12).

[20] R. Nair. «JSIM: A Java-Based Query Driven Simulation and Animation Environment». In: 1997. URL: https://www.semanticscholar.org/paper/JSIM:-A-Java-Based-Query-Driven-Simulation-and-Nair/c91905e1785dfea634cf50c8fa16503f02b700ea (cit. on p. 12).

[21] *SimJava*. URL: https://www.dcs.ed.ac.uk/home/hase/simjava/ (cit. on p. 12).

[22] Mark Little. *nmcl/JavaSim*. Java. May 2025. URL: https://github.com/nmcl/JavaSim (cit. on p. 12).

[23] Bradben. *SIM toolkit commands - Windows drivers*. en-us. URL: https://learn.microsoft.com/en-us/windows-hardware/drivers/network/sim-toolkit-commands (cit. on p. 12).

[24] R.A. Kilgore. «Silk, Java and object-oriented simulation». In: *2000 Winter Simulation Conference Proceedings (Cat. No.00CH37165)*. Vol. 1. Dec. 2000, 246–252 vol.1. DOI: 10.1109/WSC.2000.899725. URL: https://ieeexplore.ieee.org/document/899725 (cit. on p. 12).

[25] *WebGL - Low-Level 3D Graphics API Based on OpenGL*. en. July 2011. URL: https://www.khronos.org/webgl/ (cit. on p. 12).

[26] *OpenGL ES - The Standard for Embedded Accelerated 3D Graphics_2011*. en. July 2011. URL: https://www.khronos.org// (cit. on p. 12).

[27] *GPU web | WebGPU*. it-x-mtfrom-en. URL: https://developer.chrome.com/docs/web-platform/webgpu?hl=it (cit. on p. 13).

[28] Satyadhyan Chickerur, Sankalp Balannavar, Pranali Hongekar, Aditi Prerna, and Soumya Jituri. «WebGL vs. WebGPU: A Performance Analysis for Web 3.0». In: *Procedia Computer Science*. 5th International Conference on Innovative Data Communication Technologies and Application (ICIDCA 2024) 233 (Jan. 2024), pp. 919–928. ISSN: 1877-0509. DOI: 10.1016/j.procs.2024.03.281 (cit. on p. 13).

[29] Özlem Demirtaş and Bilge Kaan Görür. «A Generic Web-Based Visualization Tool for Aerospace Simulations». In: *2019 9th International Conference on Recent Advances in Space Technologies (RAST)*. June 2019, pp. 599–606. DOI: 10.1109/RAST.2019.8767779. URL: https://ieeexplore.ieee.org/document/8767779/ (cit. on p. 13).

[30] John A. Christian and Scott Cryan. «A Survey of LIDAR Technology and its Use in Spacecraft Relative Navigation». en. In: *AIAA Guidance, Navigation, and Control (GNC) Conference*. American Institute of Aeronautics and Astronautics, Aug. 2013. ISBN: 978-1-62410-224-0. DOI: 10.2514/6.2013-4641. URL: https://arc.aiaa.org/doi/10.2514/6.2013-4641 (cit. on pp. 14, 16).

[31] Matthew Berger, Joshua A. Levine, Luis Gustavo Nonato, Gabriel Taubin, and Claudio T. Silva. «A benchmark for surface reconstruction». In: *ACM Trans. Graph.* 32.2 (Apr. 2013), 20:1–20:17. ISSN: 0730-0301. DOI: 10.1145/2451236.2451246 (cit. on pp. 17, 18, 50).

[32] Michael Kazhdan and Hugues Hoppe. «Screened poisson surface reconstruction». en. In: *ACM Transactions on Graphics* 32.3 (June 2013), pp. 1–13. ISSN: 0730-0301, 1557-7368. DOI: 10.1145/2487228.2487237 (cit. on pp. 17, 47).

[33] P. Labatut, J.-P. Pons, and R. Keriven. «Robust and Efficient Surface Reconstruction From Range Data». en. In: *Computer Graphics Forum* 28.8 (Dec. 2009), pp. 2275–2290. ISSN: 0167-7055, 1467-8659. DOI: 10.1111/j.1467-8659.2009.01530.x (cit. on pp. 19, 48).

[34] F. Bernardini, J. Mittleman, H. Rushmeier, C. Silva, and G. Taubin. «The ball-pivoting algorithm for surface reconstruction». In: *IEEE Transactions on Visualization and Computer Graphics* 5.4 (1999), pp. 349–359. DOI: 10.1109/2945.817351 (cit. on pp. 19, 48).

[35] *CGAL 6.0.1 - Advancing Front Surface Reconstruction: User Manual*. URL: https://doc.cgal.org/latest/Advancing_front_surface_reconstruction/index.html (visited on 05/26/2025) (cit. on pp. 19, 48).

[36] *CGAL 6.0.1 - Scale-Space Surface Reconstruction: User Manual*. URL: https://doc.cgal.org/latest/Scale_space_reconstruction_3/index.html (visited on 05/26/2025) (cit. on pp. 20, 48).

[37] Siyu Ren, Junhui Hou, Xiaodong Chen, Hongkai Xiong, and Wenping Wang. «DDM: A Metric for Comparing 3D Shapes Using Directional Distance Fields». en. In: arXiv:2401.09736 (Apr. 2025). arXiv:2401.09736 [cs]. DOI: 10.48550/arXiv.2401.09736. URL: http://arxiv.org/abs/2401.09736 (cit. on pp. 20, 21).

[38] Yossi Rubner, Carlo Tomasi, and Leonidas J. Guibas. «The Earth Mover's Distance as a Metric for Image Retrieval». en. In: *International Journal of Computer Vision* 40.2 (Nov. 2000), pp. 99–121. ISSN: 1573-1405. DOI: 10.1023/A:1026543900054 (cit. on p. 21).

[39] H. G. Barrow, J. M. Tenenbaum, R. C. Bolles, and H. C. Wolf. «Parametric correspondence and chamfer matching: two new techniques for image matching». In: *Proceedings of the 5th international joint conference on Artificial intelligence - Volume 2*. IJCAI'77. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1977, pp. 659–663 (cit. on p. 21).

[40] Mark Jones. «3D Distance from a Point to a Triangle». In: (Jan. 1995) (cit. on p. 21).

[41] Bruno Rodriguez-Garcia, Laura Corchia, Federica Faggiano, Diana García-Tejerina, and Lucio Tommaso De Paolis. «A Tool for the Analysis and Virtual Reality Visualization of the Virtual Restoration of Cultural Heritage 3D Models». en. In: *Extended Reality*. Ed. by Lucio Tommaso De Paolis, Pasquale Arpaia, and Marco Sacco. Cham: Springer Nature Switzerland, 2024, pp. 173–184. ISBN: 978-3-031-71710-9. DOI: 10.1007/978-3-031-71710-9_13 (cit. on p. 21).

[42] Unity Technologies. *Unity - Manual: Interaction with browser scripting*. en. URL: https://docs.unity3d.com/2023.2/Documentation/Manual/webgl-interactingwithbrowserscripting.html (visited on 05/05/2025) (cit. on p. 26).

[43] *Microsoft Website | Measure runtime performance of a page using the Performance monitor tool - Microsoft Edge Developer documentation | Microsoft Learn*. en-US. URL: https://learn.microsoft.com/en-us/microsoft-edge/devtools-guide-chromium/performance-monitor/performance-monitor-tool (visited on 05/18/2025) (cit. on p. 36).

[44] Raphael Sulzer, Renaud Marlet, Bruno Vallet, and Loic Landrieu. «A Survey and Benchmark of Automatic Surface Reconstruction from Point Clouds». In: arXiv:2301.13656 (Dec. 2024). arXiv:2301.13656 [cs]. DOI: 10.48550/arXiv.2301.13656. URL: http://arxiv.org/abs/2301.13656 (cit. on pp. 46, 47, 49, 56).

[45] Amos Gropp, Lior Yariv, Niv Haim, Matan Atzmon, and Yaron Lipman. «Implicit Geometric Regularization for Learning Shapes». In: arXiv:2002.10099 (July 2020). arXiv:2002.10099 [cs]. DOI: 10.48550/arXiv.2002.10099. URL: http://arxiv.org/abs/2002.10099 (cit. on p. 47).

[46] ln-12. *ln-12/blainder-range-scanner*. Python. May 2025. URL: https://github.com/ln-12/blainder-range-scanner (cit. on p. 50).

[47] Raphael Sulzer, Loic Landrieu, Alexandre Boulch, Renaud Marlet, and Bruno Vallet. «Deep Surface Reconstruction from Point Clouds with Visibility Information». In: arXiv:2202.01810 (Feb. 2022). arXiv:2202.01810 [cs]. DOI: 10.48550/arXiv.2202.01810. URL: http://arxiv.org/abs/2202.01810 (cit. on p. 50).

[48] João Espadinha, Ivan Lebedev, Luka Lukic, and Alexandre Bernardino. «LiDAR Data Noise Models and Methodology for Sim-to-Real Domain Generalization and Adaptation in Autonomous Driving Perception». In: *2021 IEEE Intelligent Vehicles Symposium (IV)*. July 2021, pp. 797–803. DOI: 10.1109/IV48863.2021.9576034. URL: https://ieeexplore.ieee.org/document/9576034 (cit. on p. 53).

[49] Anis Farshian, Markus Götz, Gabriele Cavallaro, Charlotte Debus, Matthias Nießner, Jón Atli Benediktsson, and Achim Streit. «Deep-Learning-Based 3-D Surface Reconstruction—A Survey». In: *Proceedings of the IEEE* 111.11 (Nov. 2023), pp. 1464–1501. ISSN: 1558-2256. DOI: 10.1109/JPROC.2023.3321433 (cit. on p. 56).

[50] D.G. Lowe. «Object recognition from local scale-invariant features». In: *Proceedings of the Seventh IEEE International Conference on Computer Vision*. Vol. 2. 1999, 1150–1157 vol.2. DOI: 10.1109/ICCV.1999.790410 (cit. on p. 74).

[51] TOM McREYNOLDS and DAVID Blythe. «CHAPTER 15 - Lighting Techniques». In: *Advanced Graphics Programming Using OpenGL*. Ed. by TOM McREYNOLDS and DAVID Blythe. The Morgan Kaufmann Series in Computer Graphics. San Francisco: Morgan Kaufmann, Jan. 2005, pp. 317–359. ISBN: 978-1-55860-659-3. DOI: 10.1016/B978-155860659-3.50017-2. URL: https://www.sciencedirect.com/science/article/pii/B9781558606593500172 (cit. on p. 90).