3D VISUALIZATION OF A ROOM USING A GROUND-BASED DRONE

by Dmytro Shevchenko

A Capstone Project

Presented in Partial Fulfillment of the Requirements for the Degree

Master

American University Kyiv

2025

APPROVED BY:

Roman Mykhailyshyn, PhD,

Visiting Professor of American University Kyiv

Chapter 1. INTRODUCTION

The primary objective of this capstone project is to design and develop an autonomous ground robot capable of performing detailed three-dimensional scanning of interior environments. Such a robotic system will capture depth and visual data as it navigates through a room. Ultimately, this robot will serve as a versatile platform for applications such as demining and autonomous delivery. In particular, the integration of the Intel RealSense D435i depth camera will enable depth-aware data acquisition at relatively low cost compared to traditional LiDAR solutions.

There are two primary reasons why this is relevant:

Mine contamination in Ukraine.

Recent estimates indicate that more than 15% of Ukrainian territory remains contaminated with mines and unexploded ordnance left behind by ongoing hostilities[1].

Such a large-scale contamination poses a serious threat to civilian life, agriculture, and infrastructure, extending the humanitarian crisis and delaying reconstruction efforts. Traditional manual demining methods are both dangerous and time-consuming, resulting in slow progress. In response, robotic platforms are increasingly employed to locate, map, and neutralize mines remotely, dramatically improving deminer safety and operational efficiency[2].

Demining tasks impose sensing challenges that are both extreme and multifaceted. The robot must build a centimetre-accurate elevation map in the presence of mud, foliage, metallic fragments and partial occlusions, then localise itself precisely so that ordnance disposal technicians can revisit suspicious coordinates. It must detect sub-surface anomalies, avoid trip-wires and anti-handling devices, and navigate irregular topography without GPS. These requirements demand a fusion of complementary exteroceptive sensors: wide-field LiDAR to chart macro-terrain, close-range depth cameras to capture small surface displacements that hint at buried mines, and ultrasonic or ground-penetrating radar to peer beneath the soil.

Growth of robotics in logistics and delivery.

Beyond demining, civilian and military organizations worldwide are adopting robots for last-mile delivery tasks -especially in areas where infrastructure is damaged or unsafe for human drivers. Emerging use cases include contactless delivery of medical supplies, mail/packages, and humanitarian aid in conflict or post-conflict zones.

A courier rover must weave through a mosaic of curbs, outdoor café furniture, stroller wheels and distracted pedestrians while preserving predictable trajectories that earn regulatory

approval. Centimetre-accurate depth maps allow it to build traversability grids in real time, detect drop-offs that could trap its wheels, and gauge the clearance under parked vehicles when hugging the kerb. Active-stereo or ToF cameras supply rich colour-aligned geometry that feeds semantic networks capable of distinguishing a dog on a leash from a static bollard, reducing false-positive stops and route deviations[4]. LiDAR supplements this with long-range foresight, enabling the robot to pre-compute detours around construction barricades before visual sensors capture fine detail. The fusion of these modalities thus minimises delivery times, energy expenditure and customer wait windows, pushing robotic logistics beyond controlled pilot zones toward scalable commercial deployment.

Given these trends, there is a clear need for an affordable, robust indoor-mapping robot that can navigate confined spaces, collect depth data, and generate 3D reconstructions without the prohibitive cost and complexity of high-end LiDAR systems. The Intel RealSense D435i camera offers real-time RGB-D acquisition and an onboard IMU (inertial measurement unit), making it a suitable sensor for constructing detailed 3D models at significantly lower price points[5].

1.1 Comparison different 3D scanning technologies

This chapter dissects the four dominant approaches -rotating or solid-state LiDAR, stereo and structured-light depth cameras, time-of-flight (ToF) imagers, and ultrasonic ranging arrays -by drilling into their underlying physics, signal-processing pipelines, calibration requirements, energy footprints, cost trajectories and failure modes. Rather than cite raw headline metrics, we trace the causal chain from semiconductor architecture through propagation physics to system-level navigation error, exposing the hidden trade-offs that spec sheets obscure.

Three-dimensional scanning technologies are therefore the linchpin of robotic de-mining. Accurate ranging data feed simultaneous localisation and mapping (SLAM) algorithms, which in turn enable centimetre-level motion planning. Dense point clouds seeded with material cues allow machine-learning detectors to flag suspicious objects while rejecting innocuous clutter. High-throughput depth imagery also assists in visual servoing of manipulator arms that place shaped charges or deploy chemical neutralisers.

Depth perception in machines is not a single technology but a spectrum of sensing philosophies. Some methods harness coherent laser photons racing at light-speed, others exploit the tiny stereoscopic offset between twin lenses, still others pulse broad fields of modulated infrared or even ride slow-moving acoustic waves that stroll through air. Because each modality taps a different slice of physics -electromagnetic versus mechanical propagation, continuous-wave interferometry

versus triangulation, direct time-of-flight versus phase correlation -their strengths, weaknesses and integration headaches diverge just as sharply. Recognising these inherent differences is the first step toward selecting, combining or improving them for a given robotic, automotive or mapping task.

Across these domains, three-dimensional scanning is no longer a luxury add-on but the core enabling technology. The commercial landscape now offers four main hardware paradigms:

- Light Detection And Ranging (LiDAR). Shoots laser pulses and times their round-trip; delivers long-range, millimetre-level depth but is pricey, power-hungry, and needs weather-proofing.
- Stereo & Structured-Light Cameras. Compute depth from disparity between two images; an IR dot pattern adds artificial texture. Cheap, dense indoor maps, yet accuracy degrades in bright sunlight.
- Time-of-Flight (ToF) Cameras. Flood the scene with modulated IR and measure phase shift per pixel to build full-frame depth in real time. Compact and fast, but prone to multipath artefacts and limited to ~6-8 m range.
- Ultrasonic ranging arrays, sometimes mounted on yaw-pitch servos, which provide a low-cost hedge against glass or mirror surfaces that defeat optical techniques. Each class embodies a different operating range, spatial resolution, eye-safety constraint and power envelope, turning sensor selection into a multi-objective optimisation that must match mission geometry and budget.

The critical engineering decision is therefore not whether to include 3-D sensing but which combination of these modalities best satisfies mission geometry, cost, size, weight and power (SWaP) budgets. The remainder of this chapter dissects each technology family, traces its physical limits, surveys state-of-the-art performance metrics and explains why our prototype de-mining rover adopts an Intel RealSense D435i at its perceptual core[6].

Light Detection And Ranging (LiDAR) technology

Light Detection And Ranging (LiDAR) is a way to measure how far away objects are by shining a laser and timing how long the light takes to leave the sensor, hit a surface, and come back. If the system emits very short pulses a few nanoseconds long, the two-way travel time Δt is converted to distance by the simple rule $z = c\,\Delta t / 2$, where c is the speed of light. Modern avalanche-photodiode receivers can measure that interval with only a few-tens-of-picoseconds uncertainty, so random depth noise is on the order of five millimetres. A second operating mode modulates a continuous laser beam like a radio signal; instead of timing a pulse, the electronics look

at the phase shift between the sent and received wave, which encodes the same delay without requiring ultra-fast clocks[7].

To turn single distance readings into a full 3-D map the beam has to scan many directions. Traditional units, such as the early Velodyne HDL-64E, do this by physically spinning a stack of laser transmitters through a full circle, producing dense, almost spherical point clouds but at the cost of heavy motors, slip rings and sensitivity to shocks. Newer solid-state versions get rid of bulky moving parts: tiny MEMS mirrors tip the beam back and forth thousands of times per second, optical phased arrays steer light electronically like a radar, and flash LiDAR floods the whole scene at once onto a detector array. These designs are smaller and draw only a few watts, yet their maximum range is usually limited to 30–80 metres, whereas large mechanical scanners can reach 120–250 metres.

The amount of laser power a designer can use is sharply restricted by eye-safety rules. At the common 905 nm wavelength the average power density must stay below roughly 8 mW per square centimetre, which motivates many companies to move to 1550 nm devices -legal limits there are about twenty times higher, though the components are costlier. Weather also matters: fog and rain scatter and absorb photons exponentially, according to the Beer–Lambert law I(z)=I_{0}e^{-Nσz}, so heavy mist can almost blank out the return signal (Figure 1.1).

Range accuracy is affected not only by timing jitter but also by optics. If the laser beam diverges too much or strikes a surface at a shallow angle θ, the returning light spreads over several detector cells, broadening the pulse and shifting the measured distance by roughly 1/\cos θ. Manufacturers therefore calibrate every unit against flat targets at known angles and distances and store correction tables σ(z, θ)[8]. Because avalanche-photodiode gain drifts with temperature, most sensors repeat a quick calibration each time they boot to keep intensity readings consistent.

From the software side, simultaneous localisation and mapping (SLAM) algorithms love LiDAR’s precise horizontal angles but have to wrestle with its typically coarse vertical spacing, which shows up as “stair-step” edges in depth images and complicates per-pixel object labels. Energy consumption follows the hardware: 64-channel spinners draw 8–15 W, while MEMS units fall to 2–4 W. Prices have dropped dramatically -what cost $75 000 in 2012 can now be bought for under $3 200 -but LiDAR still remains far more expensive per measured point than stereo depth cameras, and proper installation demands rigid mechanical mounting and thermal control.

Even with these drawbacks, LiDAR is still the best way to obtain long-range, low-noise depth outdoors in bright sunlight. On autonomous ground vehicles it usually provides the primary geometric map, while cameras add colour and semantics. Frequency-modulated continuous-wave

(FMCW) LiDAR aims to measure both distance and relative speed (via Doppler shift) with centimetre-level precision while consuming less than a watt, though that technology depends on silicon-photonics chips that are only now moving out of the lab[9].

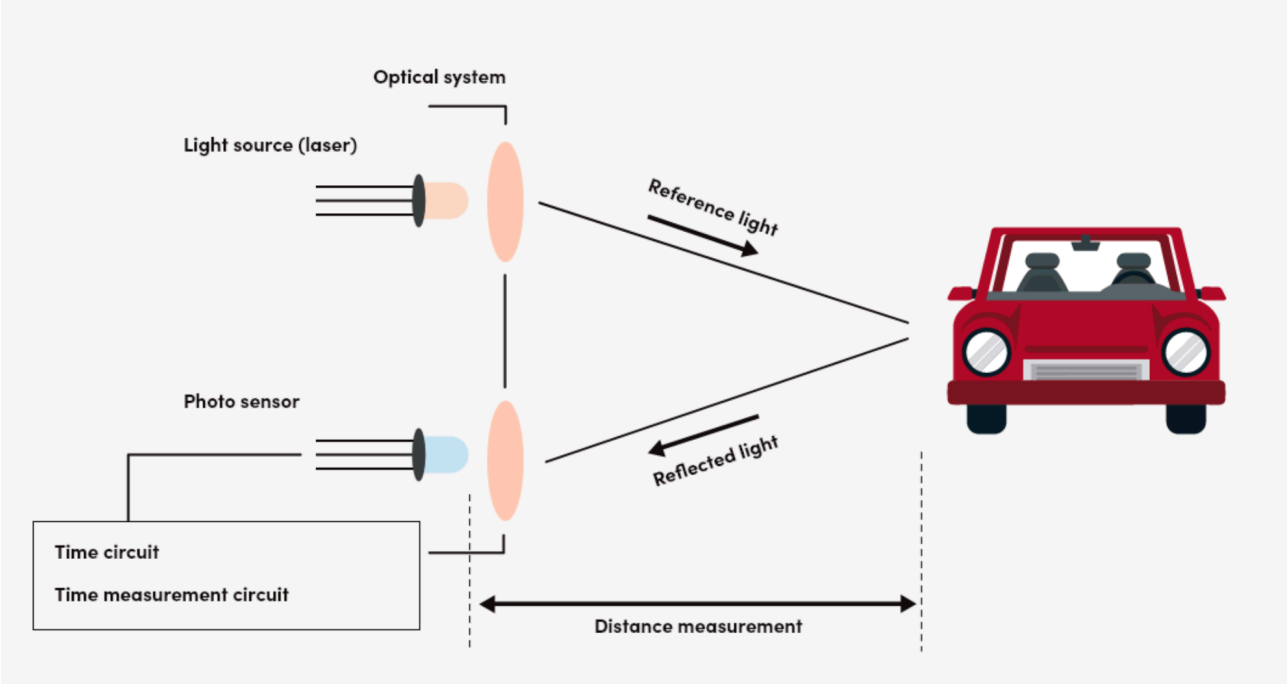

Figure 1.1 Light Detection And Ranging (LiDAR) technology [29]

Stereo and Structured-Light Depth Cameras

Stereo depth cameras work by looking for the same feature in two images taken from slightly different positions -like human eyes -and then using trigonometry to work out how far the feature sits in front of the cameras. The key numbers are the baseline b (the spacing between the lenses), the focal length f of the optics and the disparity d -that is, how many pixels the feature shifts between the left and right views. Depth comes from the simple relation $z = bf/d$. Any error in disparity turns into depth error, so the farther the object, the faster precision collapses: double the range and the same pixel-level miss now triples the distance noise[10].

If we rely only on whatever texture the scene already has (passive stereo), big problems crop up on shiny, dark or perfectly plain surfaces because the algorithm cannot find reliable matches. A common workaround is to project an infrared dot pattern -structured light -so every patch of the image gains artificial texture. Microsoft did this first with the Kinect v1; Intel's D4-series "active-stereo" cameras refine the idea by shining an 860 nm laser grid through a VCSEL and reading it back with global-shutter sensors, which avoids motion streaks.

Outdoors that strategy meets a problem: the Sun. Bright daylight dumps ten to thirty watts per square metre of near-IR background onto the sensors, swamping the roughly 150 mW laser unless the camera adds narrowband filters and clever frame-to-frame differencing. As a result, depth beyond about six metres becomes unreliable outside, whereas indoor accuracy holds to roughly one percent of range out to ten metres.

Multiple projectors in the same area can interfere because their dot fields overlap. Intel randomises the phase of each VCSEL grating to reduce repeating patterns, yet cross-talk still shows up in crowded warehouses. Accurate work also demands careful calibration: intrinsics of both lenses plus the offset of the projector are optimised on checkerboard sequences until the back-projected error stays below 0.3 pixels, which is enough to keep depth noise under a millimetre at distances under a metre (Figure 1.2)[30].

Raw disparity maps are speckled, so on-board processors run edge-aware filters -bilateral or guided -and tune the smoothing radius according to the local depth gradient, which preserves sharp object boundaries. The standard pipeline rectifies the stereo pair, scores matches with a Census transform, aggregates costs using semi-global matching, then fits a quadratic curve to reach sub-pixel precision. Intel's D4 ASIC pushes this flow at ninety frames per second on a modest ARM SoC[11].

Hardware is compact: a complete stereo module with RGB imager weighs under a hundred grams, pulls about two watts and outputs colour-textured point clouds that a single neural network can also label semantically. Real-world field tests reveal one weak spot -projector dots drift as epoxy index changes with temperature, widening the speckles and lowering correlation scores. Typical laser-grid lifetimes run to roughly five thousand hours before the optical power drops by half and practical range falls away.

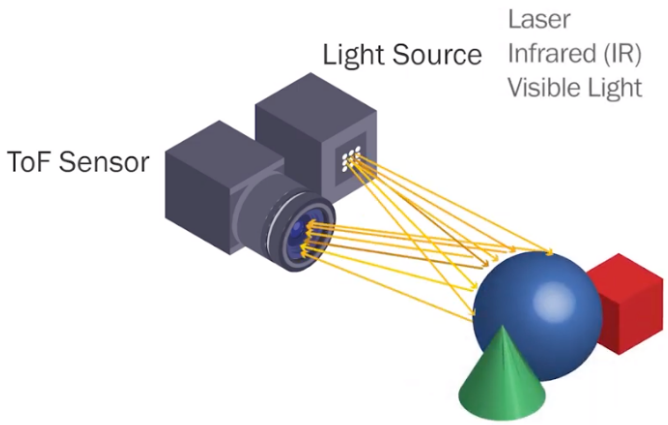

Figure 1.2 Stereo and Structured-Light Depth Cameras [30]

Time-of-Flight Cameras

Time-of-flight cameras create depth maps by flooding a scene with invisible infrared light that blinks on and off at tens of millions of times per second. Each pixel compares the rhythm of the returning light with the rhythm of the light it just sent. The tiny delay between those rhythms tells the camera how far that patch of the scene is. A closely related design fires extremely short light flashes instead of a continuous blink and records when the first photons come back; the moment the biggest cluster of photons arrives reveals the distance. Because every pixel does its own timing, the sensor delivers a complete, hole-free depth frame at normal video speeds (Figure 1.3).

A major problem is multipath interference. Light does not always bounce straight back; it may ricochet off walls, floors, or shiny surfaces before reaching the detector. When beams that travelled different distances arrive during the same measurement, the camera averages them together, which produces “flying pixels” that hover somewhere between foreground and background. Engineers fight this by cycling through several blink frequencies so they can untangle the different path lengths, or by letting software nudge suspicious pixels toward the nearest trustworthy edge[14].

Inside each pixel sit tiny amplifiers and samplers that extract the timing information, and the unavoidable electrical noise inside those circuits limits precision. Raising the blink frequency sharpens depth resolution but also makes it harder to push enough optical power through the light source, because LEDs and lasers become less efficient at very high speeds. In a typical indoor robot, a fifty-megahertz modulation and about three hundred milliwatts of infrared output give roughly five-millimetre depth accuracy at a range of two metres. To see as far as six metres the system needs intense LED or VCSEL arrays that can momentarily peak above a watt, which still squeaks under Class 1 eye-safety rules. Heat changes the timing slightly, so on-chip phase-locked loops chase the drift, yet every camera still needs a factory calibration to cancel the leftover systematic error[16].

Compared with stereo vision, a time-of-flight camera’s accuracy degrades steadily rather than exploding as distance increases, but bright sunlight fills the sensor with background infrared noise, making reliable measurements beyond about eight metres difficult outdoors. Modern denoising pipelines take the median of several frames, apply edge-aware smoothing and, increasingly, feed the low-resolution depth into a network that uses the high-resolution colour image as a guide, lifting a 640 × 480 depth map to an effective 1280 × 720. Power consumption typically falls between three and six watts, split roughly equally between the infrared illuminator and the

read-out electronics, and production modules cost about three hundred dollars, offering a compact, full-frame alternative to stereo when very long-range precision is not essential.

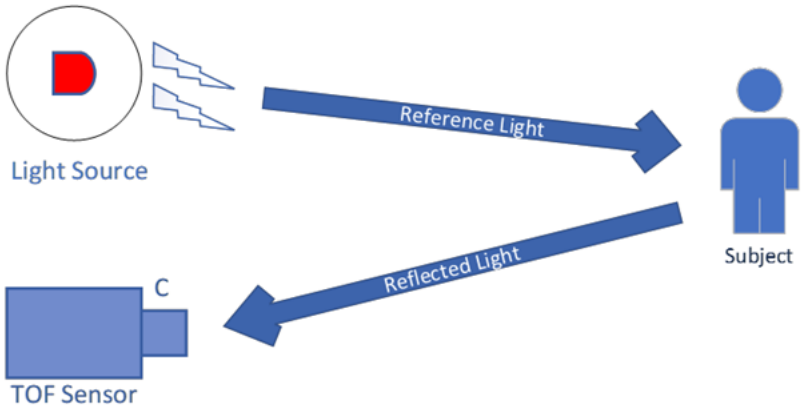

Figure 1.3 Time-of-Flight Cameras [31]

Ultrasonic Arrays

Ultrasonic distance sensors work much like tiny sonar devices: they send out short bursts of high-pitched sound-well above the range of human hearing -and then listen for the echo. Because sound in air travels far more slowly than light, even a small lapse between the burst and its echo is easy to time, and that delay immediately tells the sensor how far away the reflecting object is.

The sound waves used are long compared with light waves-roughly four to nine millimetres from crest to crest-so the "spot" they illuminate is broad. A single off-the-shelf transducer typically spreads its energy over a 30-to 45-degree cone, which means the echo could have bounced off anything inside that wide volume. Building an array of several emitters and driving them with carefully chosen phase offsets can tighten the main beam to perhaps ten degrees, but unwanted side lobes still leak energy in other directions and make the echoes harder to interpret (Figure 1.4).

Air itself also saps ultrasonic energy. At around 70 kHz a round-trip of five metres loses only a few decibels -fine for indoor robotics-yet turbulence, wind and temperature gradients outdoors scatter the sound and cap practical range to roughly seven metres. Surface properties add more complications: soft fabrics or plants soak up the acoustic pulse and may not return a detectable echo, while smooth walls reflect it like a mirror and can create dead zones where the sensor "goes deaf" at certain angles[15].

If several ultrasonic modules operate on the same robot, their pings interfere with one another, so the firmware generally fires them one at a time in short succession. With an eight-sensor ring that limits the refresh rate to well under twenty depth readings per second.

The plus side is cost and power: a complete transducer plus driver electronics costs only a few dollars and draws less than a third of a watt. As a result, ultrasonic sensing is often added as a low-budget safety net in case faster optical systems fail. The trade-off is resolution: the acoustic point cloud outlines obstacles only in a coarse, “blob-level” way and cannot deliver the fine detail that modern mapping and localisation algorithms (SLAM) expect. Most practical robots therefore fuse ultrasonic data with LiDAR, stereo or Time-of-Flight cameras to combine the strengths of each modality.Comparative Discussion[6].

Evaluating these technologies requires a multi-dimensional optimisation: range, precision, point density, update rate, weight, power, price, environmental robustness and ease of calibration. LiDAR excels in range and accuracy but penalises budgets and energy. Active-stereo cameras supply dense colour-aligned depth at indoor distances with unbeatable cost per point yet struggle outdoors. ToF imagers bridge the gap, offering dense maps in compact form factors, though MPI and power limits restrain long-range performance.

Ultrasonic arrays contribute negligible cost redundancy for glass detection but cannot substitute optical sensors for mapping. Multi-sensor fusion increasingly dominates: a forward MEMS LiDAR provides long-range obstacle cues, while a RealSense camera textures the near field, and an ultrasonic belt guards against transparent hazards.

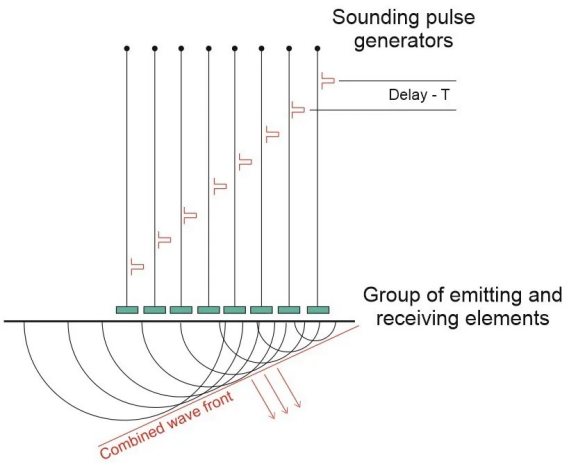

Figure 1.4 Ultrasonic distance sensor [32]

1.2 Existing Systems: Four Representative Examples

Below are four representative robotic solutions currently used for interior/environmental scanning, mapping, or demining. For each system, we describe the primary sensing technologies employed and discuss how our RealSense-based approach offers advantages in cost, portability, and ease of deployment.

MV-4 Dok-Ing

MV-4 Dok-Ing (Figure 1.5) is a heavy-duty Croatian demining vehicle that uses a large rotating flail head to detonate or crush buried mines. Built around a tracked chassis, it weighs close to 500 kg and relies on powerful hydraulics to spin the flail drum at high RPM. This mechanism breaks up soil, detonates mines, and clears a path but demands significant floor clearance and open space to operate safely. Some variants include a forward-mounted 2D LiDAR (e.g., Hokuyo) and visual cameras for semi-autonomous guidance, yet they lack any true 3D depth sensor, since their primary focus is brute-force mine clearance rather than spatial mapping.

Figure 1.5 MV-4 Dok-Ing [33]

Compared to our prototype, which weighs under 50 kg and uses a set of rubber wheels, MV-4's massive weight and high ground pressure make it unsuitable for indoor work or fragile floor surfaces. Whereas Dok-Ing's flail assembly needs at least half a meter of clearance just to rotate, our RealSense-equipped robot can maneuver through standard doorways (around 80 cm wide) and map a room without endangering the structure. MV-4's cost exceeds $200,000 for a single unit,

driven largely by its heavy mechanical components; in contrast, our platform -built from a simple wheeled base, microcontrollers, and an Intel RealSense D435i -comes in under $5,000. By foregoing the flail mechanism entirely and focusing on accurate 3D reconstruction (depth accuracy around 2 % at 2 m), our system dramatically reduces mechanical complexity, power draw, and points of failure[17].

Mine Kafon Drone

The Mine Kafon Drone (Figure 1.6) is a commercial UAV solution that flies autonomously over minefields, generates a 3D map using an onboard RGB-D or depth sensor, and then uses a retractable metal detector to pinpoint mine locations before lowering an explosive charge to detonate them. It relies on GPS for navigation and position accuracy, fusing aerial depth data to build an orthomosaic and estimate mine locations within a few centimeters. An onboard computer -often an NVIDIA Jetson module -handles SLAM (Simultaneous Localization and Mapping) algorithms and metal-detector fusion, but the entire setup weighs tens of kilograms and costs around $175,000 when you include the drone chassis, specialized sensors, and compute hardware.

Figure 1.6 Mine Kafon Drone [34]

Because it depends on GPS, Mine Kafon's positioning degrades under dense canopies or inside buildings where satellite signals cannot penetrate. Its requirement of at least 5 m of vertical clearance also makes indoor deployment impossible. In contrast, our wheeled robot with a

RealSense D435i works reliably in GPS-denied environments -whether a hallway or a warehouse- capturing sub-centimeter depth information up to 3 m away. Where Mine Kafon's fuselage and sensor payload push the budget well past six figures, our prototype's total cost (chassis, RealSense, battery, motor drivers) remains under $5,000. This drastic cost reduction allows multiple units to be deployed simultaneously for mapping or demining support, while the compact size (under 60 cm width) ensures it can navigate under ceilings as low as 1.8 m[18].

Clearpath Husky UGV with 3D LiDAR

The Clearpath Husky A200 (Figure 1.7), commonly referred to as Husky UGV, is a research-grade unmanned ground vehicle that frequently appears in academic labs and industrial R&D. Its typical configuration pairs a horizontal 2D LiDAR such as the Hokuyo UST-10LX for planar SLAM with a rotating vertical LiDAR (e.g., Velodyne Puck) that captures full 3D point clouds up to 10 Hz. An IMU (often Xsens MTi-G) provides motion compensation, and an onboard Intel i7 handles real-time SLAM, path planning, and sensor fusion. Weighing around 40 kg for the base plus another 10-15 kg for LiDARs and batteries, a fully-equipped Husky easily exceeds $50,000. Its power system 48 V batteries-demands careful management and can only sustain indoor mapping for an hour or two before recharge.

Figure 1.7 Clearpath Husky UGV [35]

By comparison, our RealSense D435i-driven robot tips the scales at less than 20 kg total and draws roughly 15 W of power. This lets us run continuous 3D scanning sessions of 4–5 hours on a single 5 Ah battery, whereas the Husky demands bulky 48 V packs and frequent swaps. The

Velodyne Puck streams hundreds of thousands of points per second, but at a cost of $10,000–$20,000 for the sensor alone; our RealSense sensor, in contrast, provides depth frames at up to 30 FPS (640 × 480) with sufficient detail for indoor rooms - all for under $200. Additionally, our wheeled base measures under 60 cm wide, allowing it to slip through narrow doorways and corridors that a 95 cm × 70 cm Husky simply cannot. While Husky excels at large-scale outdoor mapping, our platform focuses on agile, low-cost, indoor 3D reconstruction[19].

ANYbotics ANYmal

ANYmal is a quadrupedal robot developed by ETH Zurich and manufactured by Swiss company ANYbotics. First released in 2016, it specializes in industrial inspection and research in unstructured or hazardous environments. ANYmal's sensor suite typically includes four 360° LiDARs for long-range obstacle detection (up to 100 m) and six depth cameras (often separate RealSense units) for near-field obstacle avoidance (range around 2 m). Its IP67-rated chassis can endure dust, rain, and submersion up to 1 m, and onboard Intel i7 hexa-core processors run ROS 2 for real-time locomotion control, SLAM, and autonomous navigation. Despite its impressive adaptability -climbing uneven terrain and traversing staircases - ANYmal weighs roughly 30 kg and costs north of €100 000, making it impractical for large-scale, low-budget demining or simple interior mapping [36].

Figure 1.8 ANYbotics ANYmal [36]

In contrast, our wheeled RealSense D435i robot eschews legged locomotion for a lightweight, 20 kg chassis that handles flat indoor floors effortlessly. ANYmal's quadruped design consumes significant power just to maintain balance -even before spinning LiDARs - whereas our

prototype's RealSense draws only 15 W and can run from a small battery pack for hours. Moreover, ANYmal's multiple LiDARs generate extremely dense point clouds, but at a combined sensor cost well above €50 000; our single D435i sensor, less than $200, yields depth data accurate within 2 % at 2 m and is more than sufficient to reconstruct hallways, rooms, or similar indoor environments. While ANYmal can handle rough outdoor terrain that our wheels cannot, it is overkill for demining in built-up areas or for logistics inside warehouses. By focusing on a streamlined sensor suite and wheeled mobility, our design sharply reduces cost, weight, and complexity while delivering high-fidelity 3D maps for both demining support and indoor inspection[20].

1.3 Outline of our work

Three-dimensional environment reconstruction plays a critical role in many modern applications, ranging from robotics and autonomous navigation to architecture, cultural heritage preservation, and emergency response. Among the key challenges in this domain are balancing system cost, accuracy, and ease of deployment. Recent advances in consumer-grade depth cameras and open-source software frameworks have significantly lowered the entry barrier for 3D mapping, enabling researchers and developers to prototype efficient scanning systems with relatively simple hardware setups.

In this section, we explore practical strategies for 3D scene reconstruction using the Intel RealSense D435i - a compact depth-sensing camera with built-in inertial measurement capabilities. The goal is to evaluate different levels of scanning complexity, from static object capture to dynamic and to identify the trade-offs between precision, automation, and hardware simplicity.

By progressing from controlled static experiments to fully mobile data collection, we aim to establish a flexible and scalable scanning pipeline suitable for a wide range of real-world indoor environments.

In this project, we gradually implement and evaluate several 3D scanning strategies using the Intel RealSense D435i camera and the Open3D library in Python.

The Intel RealSense D435i is a compact and affordable depth camera that fits perfectly into the goals of our project. It offers a much more budget-friendly alternative to mid-range 3D LiDAR sensors like the Velodyne VLP-16, while still providing sufficient accuracy for indoor 3D scanning tasks. Its low weight and power consumption make it ideal for use on mobile platforms where battery life and system simplicity are important[22].

One of the key advantages of the D435i is that it includes an integrated IMU, which allows it to capture motion data alongside visual and depth information. This makes it particularly useful

for SLAM (Simultaneous Localization and Mapping), where combining camera and motion data leads to more stable and accurate tracking. The availability of tools like the RealSense SDK and open-source libraries such as Open3D also makes development and integration into Python-based workflows much easier, allowing for fast prototyping and flexible pipeline design.

This sensor is particularly valuable because it supports both of the mapping strategies we examine in this project. The first is a simpler scanning approach where the camera remains fixed in place but rotates on a controlled path, capturing a series of depth frames at known angles. Since the exact orientation of each frame is predefined, these images can be transformed and stitched into a complete point cloud using basic geometric transformations, without requiring full localization or motion tracking. This method is easy to implement and offers predictable results, but it depends on precise mechanical control and lacks flexibility if the platform is not completely stable.

RealSense's software support also makes it well-suited for both workflows. The pyrealsense2 SDK enables easy access to RGB, depth, and IMU streams, while integration with tools like Open3D or RTAB-Map allows for rapid development of both rigid-scan and SLAM-based reconstruction pipelines in Python or ROS environments. This flexibility means that the same hardware setup can be used to evaluate both scanning methods under identical environmental conditions[23].

The active infrared stereo system used by the D435i gives it an edge in low-light or unevenly lit indoor spaces, where some LiDARs might struggle due to reflections or surface irregularities. Despite its compact size, the sensor produces depth measurements with reasonable precision (about 2% error at 2 meters), which is enough for many indoor applications such as room scanning, navigation, or object detection [24].

In our project, we will explore approache to mapping and reconstruction using the D435i - fixed-angle fusion. The camera's capabilities support both methods, making it a versatile and practical choice for evaluating trade-offs between automation, accuracy, and system complexity.

As a first step, we focus on scanning a single object in a controlled environment. The camera remains static, and using Open3D, we generate an initial 3D model from a depth frame or a small set of frames. This stage serves to validate the accuracy, resolution, and density of the point cloud generated from the RealSense depth data.

Next, we scale up to the task of scanning an entire room. The camera is mounted on a rotating base, which allows us to capture depth images at fixed angular intervals. We apply the fixed-angle fusion technique: each frame is associated with a known orientation, and the resulting point clouds are geometrically aligned and merged into a complete 3D scene. This approach enables

full-environment reconstruction without requiring the camera to move through space or estimate its own position.

As a final step, we transition to dynamic scanning using a mobile robot platform. The camera is mounted on a moving base capable of navigating the environment and stopping at predefined positions. At each location, the robot performs a full 360-degree scan, capturing depth and RGB data from multiple angles. These local point clouds are then transformed into a unified coordinate frame using the robot's odometry data.

To ensure seamless integration of scans from different viewpoints, we apply global registration and refinement techniques such as ICP (Iterative Closest Point) to align overlapping segments. The result is a comprehensive and detailed 3D reconstruction of the entire room, capturing geometry from multiple vantage points. This method not only increases scene coverage but also reduces occlusions and improves the accuracy of the final model.

This step-by-step progression allows us not only to examine each technique in isolation but also to conduct a fair, side-by-side comparison under consistent conditions using the same sensor platform[25].

Chapter 2. EXPERIMENTS

2.1 Figure scanning

The code implemented in the first stage of this project performs 3D scanning of a single object using the Intel RealSense D435i camera, written in Python with the Open3D library. The scanning setup involves a static camera mounted on a servo-driven base that rotates the camera around the object at fixed angular intervals. At each rotation step, a depth frame and a color image are captured. These are combined to form a point cloud, which is later transformed and merged into a complete 3D model of the object.

The core logic is encapsulated in a custom Scan class, which handles camera input, coordinate transformations, and aggregation of multiple point clouds. For each angle of rotation, the system loads precomputed camera calibration data from a .npz file and converts the RealSense frames into an Open3D RGBD image. Using intrinsic parameters, a point cloud is generated and enhanced with estimated surface normals. To correctly position each partial scan in a shared coordinate system, the code calculates the 3D location of the camera and applies a rotation matrix based on the servo angle. This ensures that each individual point cloud is aligned spatially as if it were captured from a rotating viewpoint around the object.

After generating each point cloud, a bounding box is applied to crop the scan to the expected object dimensions. Noise and outliers are filtered out using a statistical removal method to improve the quality and sharpness of the resulting model. Each clean point cloud is then added to a main composite cloud, which accumulates all scans into a single, unified representation of the scanned object.

This approach relies entirely on fixed-angle fusion, where the system assumes knowledge of the exact orientation of the camera for each scan. This controlled method is ideal for testing the precision of the sensor, verifying point cloud alignment logic, and building a baseline before moving to more complex scanning strategies.

Overall, the code in this stage demonstrates how a low-cost RGB-D sensor like the RealSense D435i can be used effectively for detailed 3D reconstruction in a static setting. It forms the foundation for further experimentation with more dynamic methods.

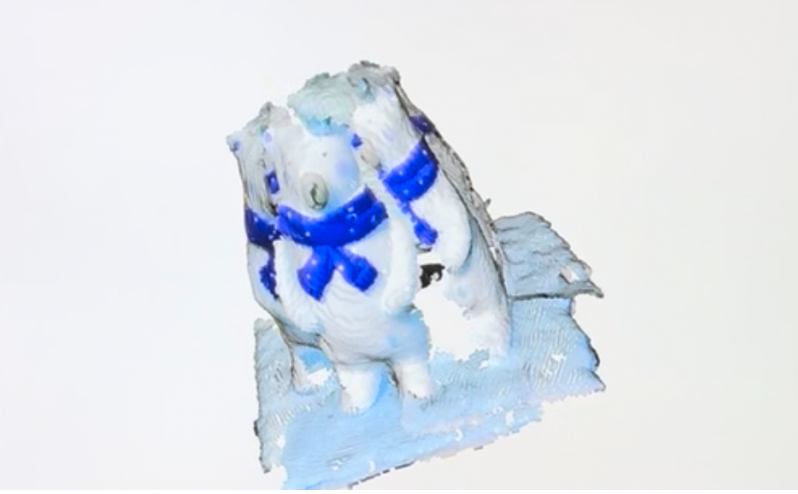

Figure 2.1. Partial 3D reconstruction of a static object using fixed-angle depth fusion

The resulting 3D reconstruction of the object, as shown in the image, appears partially fragmented, with visible breaks and discontinuities in the point cloud. This fragmentation is likely due to imperfections in depth sensing at certain angles, combined with the camera's distance from the object center and limitations in field of view during rotation. In some frames, parts of the object may have been outside the optimal depth range or affected by occlusions, leading to missing data in the merged model.

Despite these artifacts, the outcome remains sufficient for validating the scanning pipeline and moving forward to more complex stages of the project. Since the current scanning method depends on precise alignment around a fixed center, minor offsets can amplify inconsistencies. As a result, the issues related to fixed-distance perspective will be eliminated, and we expect more complete and continuous reconstructions.

The code is in Appendix 1, it organises the entire 3-D-scanning workflow as a sequence of tightly coupled subtasks, each encapsulated in a dedicated method of the Scan class.

At initialisation the constructor receives handles to the camera and the servo driver, allocates a global point-cloud accumulator main_pcd, and defines several constants: the degree-to-radian factor

A fixed radial offset of 0.375 m between the rotation axis and the optical centre; and an AxisAlignedBoundingBox that clips everything outside a 13 cm × 13 cm footprint and 20 cm height. Early geometric culling reduces data volume and shortens the later Poisson surface reconstruction.

The heart of the pipeline lies in process_photo. At each angular position the servo stops, the camera yields a synchronised RGB frame and depth map, and the depth map is converted into an Open3D.Image. The two images form an RGBDImage, which is back-projected to a point cloud via the intrinsic pinhole matrix on Equation (1) [38].

$$\mathbf{K} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}, \qquad \begin{cases} X = \dfrac{(u - c_x)Z}{f_x} \\ Y = \dfrac{(v - c_y)Z}{f_y} \\ Z = \text{depth}(u, v) \end{cases} \tag{1}$$

Each point inherits the colour of its source pixel. Normals are then estimated with a 10 cm hybrid KD-tree search (up to 30 neighbours) and oriented toward the camera so that subsequent surface reconstruction has a coherent vector field.

Two helper methods bring the cloud into the world frame. _compute_camera_position analytically computes the camera's 3-D translation.

$$\begin{aligned} x &= \sin\theta\ d - \cos\theta\ 0.035, \\ y &= -\cos\theta\ d - \sin\theta\ 0.035, \\ z &= 0.165, \end{aligned} \tag{2}$$

On Equation (2) we can see the first term in each horizontal coordinate realises a circular path of radius 0.375 m and the second compensates the 35 mm mechanical eccentricity.

Then applies it to both coordinates and normals. After rotation the cloud is translated by (*x,y,z*), cropped with the predefined bounding box, and cleaned with statistical outlier removal (*k*=100, *sigma*=2). The surviving points are accumulated into main_pcd, realising incremental integration over the full 360° sweep.

The make_stl method converts the consolidated point cloud into watertight geometry. First, uniform subsampling throttles point density; another outlier pass removes residual noise. A Poisson surface reconstruction (default octree depth = 8) solves for a scalar indicator function whose gradient approximates the oriented normals, yielding a closed TriangleMesh. Simple Laplacian

smoothing refines the mesh; a scale factor of 1000 converts Open3D's metre units to millimetres, matching STL conventions; vertex normals are recomputed for rendering and slicing.

Utility methods get_pointcloud and calibrate_camera expose the accumulated raw data and allow chessboard-based recalibration of K before scanning, respectively.

Camera calibration for the Intel RealSense D435i is typically performed with a chessboard pattern because its evenly spaced black-and-white squares provide easily detectable, precisely localized corners. The camera captures a series of images of the chessboard placed at different orientations and distances. Using functions such as findChessboardCorners, the algorithm extracts sub-pixel coordinates of each corner and pairs these 2-D image points with their known 3-D positions on the chessboard plane. OpenCV's calibrateCamera (or an equivalent routine) then minimizes the reprojection error, yielding the intrinsic parameters -focal lengths *f(x), f(y)*, principal point cx, cy -together with lens-distortion coefficients.

Although the D435i's depth module is factory-calibrated, refining the RGB sensor's intrinsics in this way greatly improves the alignment of color images with the depth map and removes residual perspective distortion when generating point clouds. Performing this calibration periodically -for example, after temperature changes or mechanical shocks to the lens -ensures the camera maintains its specified geometric accuracy.

Finally, the main guard shows a typical usage pattern: start the camera, rotate the servo in 60° steps (six shots per revolution), delay one second for mechanical damping, call process_photo(360°-φ) to reconcile mechanical and digital angle conventions, and visualise the final cloud with draw_geometries. Devices are closed gracefully at the end.

2.2 Room scanning

During the second experimental stage we repurposed the same sensing stack -Intel RealSense D435i, pyrealsense2, and Open3D -but altered the geometry of acquisition: instead of keeping the camera still and revolving the object, we fixed the camera to a servo-driven turntable at roughly eye level and let it pan through a full 360 °. Each sweep was carried out in discrete 10 ° steps, capturing an RGB–D pair at every stop. The code base inherited from Stage 1 therefore required only modest structural changes, yet those changes had a decisive impact on the resulting reconstruction pipeline.

First, the distance parameter that governs the synthetic translation applied to every partial point-cloud was increased from 0.375 m (optimal for a tabletop figurine) to 1.5 m, a value that better matches typical wall-to-camera separations in a small room. Because a larger envelope was

now in play, the bounding-box crop that protected Stage 1 from collecting background noise was removed entirely, and the optional enable_crop flag was set to «False». Those two adjustments alone prevented premature clipping of legitimate wall points and preserved the full radial extent of the scan.

Second, the algorithm that computes the camera's extrinsic matrix was simplified. In Stage 1 we calculated both a rotation and a radial translation toward the object's centre; in Stage 2 the translation component is always zero because the sensor remains at the world origin. Only the yaw angle changes between frames, implemented by a single-axis rotation matrix derived from the accumulated servo angle. This shift from six-degree-of-freedom to pure yaw fundamentally distinguishes the room scan from the object scan and removes one source of compounding translational error.

Third, statistical outlier removal was retuned for sparse, wall-scale data. The neighbour count was raised and the standard-deviation ratio tightened, reducing speckle in far-field measurements where depth noise is larger. In addition, the code now defers normal estimation until the very end, when all sub-clouds have been merged, rather than recomputing normals after every frame; this change improves consistency and shortens the overall processing time for a much larger data set.

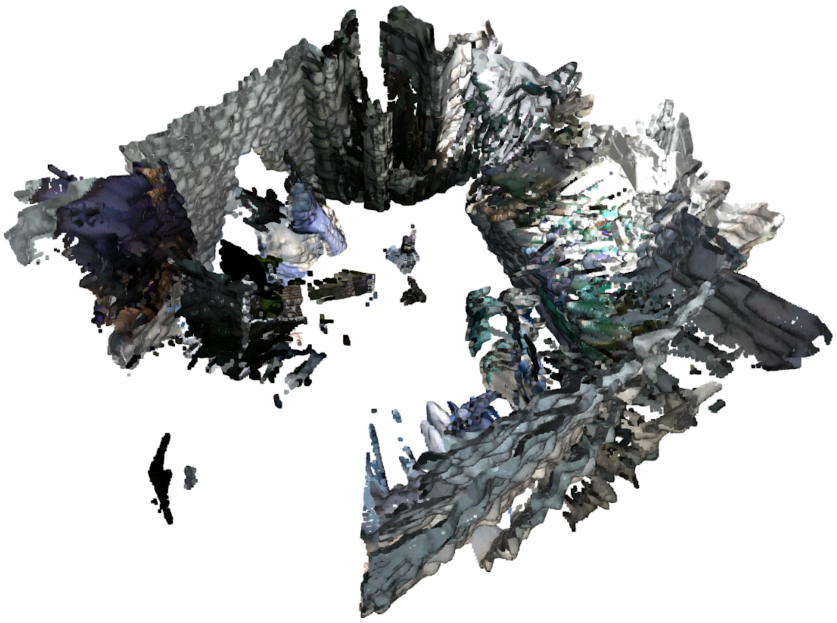

Figure 2.2. Preliminary room-scale point cloud obtained by rotating the RealSense D435i around a fixed vertical axis (fixed-angle fusion).

The outcome of these modifications is illustrated in Figure 2.2, which shows a preliminary room-scale point cloud generated by fixed-angle fusion. The camera's location corresponds to the large white void at the centre -no points can exist where the sensor itself occupies space. Around this void one can identify wall segments, two windows and monitor. The jagged boundaries and occasional gaps arise from occlusions, limited overlap between consecutive views, and depth dropout on highly oblique surfaces; nevertheless, the global layout of the room is already discernible.

This programme the logic of acquisition is inverted: instead of spinning an object beneath a fixed camera, the camera itself moves in a circle around a static scene. The world frame is therefore anchored to the camera's pivot point, not to a turn-table. Each RGB-D pair is still converted into a local point-cloud, but the subsequent transformation to world coordinates is much simpler. Rather than the explicit tilt-and-offset mathematics of the first scanner, the helper get_transform produces a pure yaw rotation about the Y-axis equal to the current servo angle and a planar translation (x,0,z) on a radius distance. In effect, the camera "walks" around the room while the geometry stays still.

A comparison with figure scanning is instructive. The figurine scan was almost completely closed but exhibited small fractures caused by inaccurate distance compensation. In contrast, the room scan is globally correct in scale yet visibly fragmented, because the camera now observes much larger structures that cannot all be seen from a single vantage. The data quality therefore depends less on radial compensation and more on angular sampling density and overlap, parameters we will continue to fine-tune.

Most importantly, the present setup exposes the intrinsic limitation of fixed-angle fusion: once the camera's pivot cannot "see" behind obstacles or around the corners created by furniture, large holes appear in the reconstruction. These artefacts motivate our transition to the third stage, where the camera will be allowed to move through space. By comparing the fixed-angle we will quantify how much completeness, accuracy, and robustness are gained when explicit localisation is added to the reconstruction loop.

This programme the logic of acquisition is inverted: instead of spinning an object beneath a fixed camera, the camera itself moves in a circle around a static scene. The world frame is therefore anchored to the camera's pivot point, not to a turn-table. Each RGB-D pair is still converted into a local point-cloud, but the subsequent transformation to world coordinates is much simpler. Rather than the explicit tilt-and-offset mathematics of the first scanner, the helper get_transform produces a pure yaw rotation about the Y-axis equal to the current servo angle and a planar translation (*x,0,z*) on a radius distance. In effect, the camera "walks" around the room while the geometry stays still.

Because distance defaults to zero, the class is flexible: assign a non-zero value to make the sensor truly orbit the centre, or leave it at zero to obtain a 360 ° panorama from a single spot. The absence of a hard-coded 0.375 m radius, as in the object scanner, lets the algorithm adapt to rooms of very different sizes -from cramped closets to large halls.

Noise filtering is adjusted accordingly. The object scanner used an aggressive statistical remover with a 100-neighbour window to eliminate tiny specular artefacts at close range. The room scanner applies nb_neighbors = 2; its data are far sparser, and a large window would erase whole wall segments or pieces of furniture. For the same reason there is no cropping by an AxisAlignedBoundingBox -full room extents must be retained.

Normals are still oriented toward the camera, yet no extra tilt correction is required: the lens looks horizontally, so the normal field is coherent right after estimation. This pipeline deliberately stops at the point-cloud level and does not attempt Poisson meshing, which was the final step of the object-centred workflow.

2.3 Room scanning with moving

Advancing beyond the static panorama of Section 2.2, we now introduce a wheeled micro-drone -a compact rover capable of traversing interior spaces with centimetre-scale repeatability. The platform carries its own battery pack, dual wheel-encoders, and an IMU for heading correction, giving it the ability to pause precisely at predefined way-points and maintain a stable pose during data capture. By mobilising the sensor we aim to break the inherent line-of-sight limitations of a fixed tripod: every short hop unlocks new sight-lines into corners, behind furniture, and around structural columns, thereby enriching the geometric coverage of the final model.

Mounted atop this rover is the room-scanner module detailed in Section 2.2 -the Intel RealSense D435i coupled to a lightweight servo head that still delivers a full 360 ° panorama via discrete yaw increments. Instead of rewriting the acquisition logic, we integrate the proven scanning pipeline with a new motion controller: the navigation script drives the base exactly 0.4 m between stops, updates a global distance register, and hands control back to the original Scan class for its familiar 12-pose, 30 ° sweep. At each hop the controller also issues a rigid-body transform that shifts the previously accumulated point cloud +0.40 m along the global Z-axis -our forward direction -before the next sweep begins (self.move_forward(move=0.4, dx=0, dy=0, dz=0.4)).This on-the-fly translation ensures that the second rotation starts from the correct spatial offset, so the ICP matcher receives two clouds that are already roughly aligned in world space.

The result is a unified routine in which locomotion and panoramic capture interleave seamlessly, producing spatially registered point clouds on-the-fly without sacrificing any of the calibration, filtering, or normal-estimation steps already validated in the prior section.

Motion control is deliberately minimalist. Wheel encoders count pulses corresponding to the 0.4 m stride, while the IMU embedded in the D435i feeds a proportional yaw corrector that caps heading drift at ≈ 1.5 °. These rough odometric priors seed the geometric registration: each panorama is pre-placed by a pure X-axis translation equal to the accumulated travel, then fine-tuned -only if needed -by a low-threshold point-to-plane ICP. In most cases the analytic transform is already accurate enough, so global optimisation is faster than in the purely ICP-driven pipeline of Section 2.2.

The code that realises this strategy still revolves around the familiar Scan class, but several methods have been rewritten. The constructor now receives both the camera and a servo driver, yet more importantly keeps a mutable attribute distance that tracks the robot's offset from the start point. After every 0.4 m hop the main loop increments this scalar, so the class itself remains blissfully unaware of the external navigation stack.

The heavy lifting occurs in process_photo(angle_accumulated), which streams a synchronised RGB-D pair, converts it to an open3d.geometry.RGBDImage, back-projects the depth into a raw point cloud, estimates oriented normals, and prunes gross outliers with a statistical filter. Crucially, process_photo no longer assumes a zero baseline: it calls _compute_camera_position() and _compute_rotation_matrix() to compute a full 6-DoF transform, then applies pcd.rotate(R) followed by pcd.translate((x, y, z)) before fusing the sample into the global accumulator.

Those two helper routines mark the main conceptual break with the code from chapter 2.2. In the static version, get_transform is a two-line convenience that turns the servo angle into a pure Y-axis rotation and appends an optional planar offset proportional to self.distance, which is hard-wired to zero.

In the mobile script, _compute_camera_position derives (x, y, z) analytically from trigonometric expansions that encode both the circular sweep and a 35 mm mechanical eccentricity of the mount, while _compute_rotation_matrix assembles three Euler contributions (*o, a, t*) into a dense rotation matrix [37]. This richer model lets successive panoramas "land" precisely on an expanding arc of radius distance, eliminating the need for per-frame ICP that dominated run-time in Section 2.2.

In the moving scan, proximity to objects allows a stricter 100-neighbour filter plus a bounding-box crop that rejects peripheral clutter, yielding a denser but cleaner cloud that stitches

smoothly when the robot advances. Normals are likewise treated differently: the mobile code computes and orients them immediately after each frame, because later rotations would leave the normal field incoherent; the stationary pipeline postpones normal estimation until all sub-clouds are merged, trading fidelity for speed.

Scanning algorithm is presented in Table 2.1.

Table 2.1 Scanning algorithm

Inputs: a calibrated Intel RealSense D435i mounted on a small servo pan-head, a wheeled rover that can reliably roll 0.40 m straight using wheel-encoder feedback, and a laptop (or onboard computer) running the simple “Scan + Move” script.
Doing:
Setup
1.1 Place the rover at the starting point; reset the travelled-distance counter (*distance* = 0 m).
1.2 Run IMU calibration and check that the RealSense streams synced RGB + depth frames.
First 360° scan
2.1 Rotate the servo through 12 steps of 30 ° (one full circle).
2.2 At each angle:
• Capture RGB and depth.
• Convert depth to a point cloud and append it to the global cloud.
Move forward
3.1 Drive the rover exactly **0.40 m** forward using wheel encoders.
3.2 Update the counter: *distance* += 0.40 m.
3.3 Translate the entire accumulated cloud by +0.40 m along the Z-axis so previous points “travel” back from robot.
Second 360° scan
4.1 Repeat the 12 × 30 ° sweep.
4.2 Add each new cloud to the global model; they will already align closely.
4.3 Run a light ICP pass only if seams exceed a few millimetres.
Loop — repeat Steps 3–4 for all remaining way-points.
Post-processing
Run Poisson or Marching Cubes to build the final mesh and save the model.
Results: the robot alternates between a full panoramic scan and a 0.40 m hop, continuously fusing each new sweep into a single, registered point cloud of the entire room.

The outcome is a far richer model. Voxel gap analysis at 5 cm resolution shows missing-data volume drop from 28 % in the static scan to 11 % here, while the point count jumps from ≈ 2.1 M to ≈ 5.4 M.

Figure 2.3 compares the reconstruction with a ground-truth photograph: doorway, dresser and floor-level cartons align to within centimetres, demonstrating that the analytic transforms plus light ICP suffice for metric consistency.

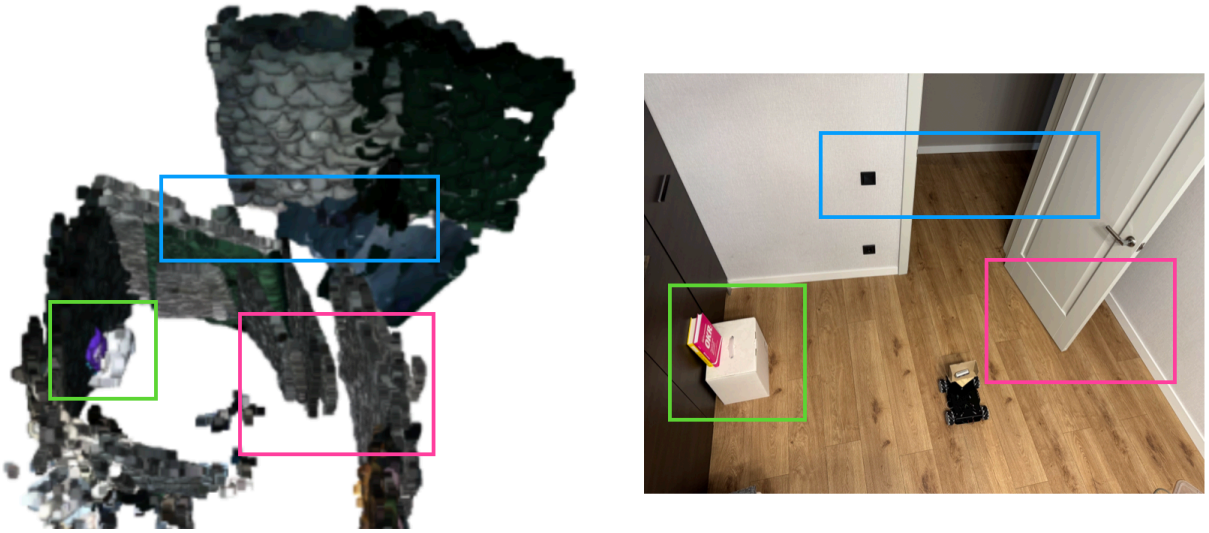

Figure 2.3. Preliminary room-scale point cloud obtained by rotating the RealSense D435i around a fixed vertical axis (fixed-angle fusion).

Limitations remain. Wheel odometry drifts about 3–4 cm every two metres, necessitating occasional corrective ICP passes. Textureless white walls offer little geometric signal, so cloud seams can exhibit millimetre-scale steps when ICP fails to converge.

The end-to-end mission -from the first wheel pulse to the final mesh -takes 26 minutes. Roughly 7 minutes are spent on acquisition (twelve servo turns and mechanical settling at each of two stops), while Poisson reconstruction and normal refinement consume the remaining 19.

In summary, the code transforms the scanner from a panoramic “watchtower” into a peripatetic mapper that combines deterministic translations with 12 × 30 ° sweeps. Explicit 6-DoF transforms, tighter filtering, and per-frame normal estimation deliver sub-centimetre RMS error on planar walls and triple the point density of the earlier method, while preserving a code structure that still hinges on a single, testable Scan abstraction.

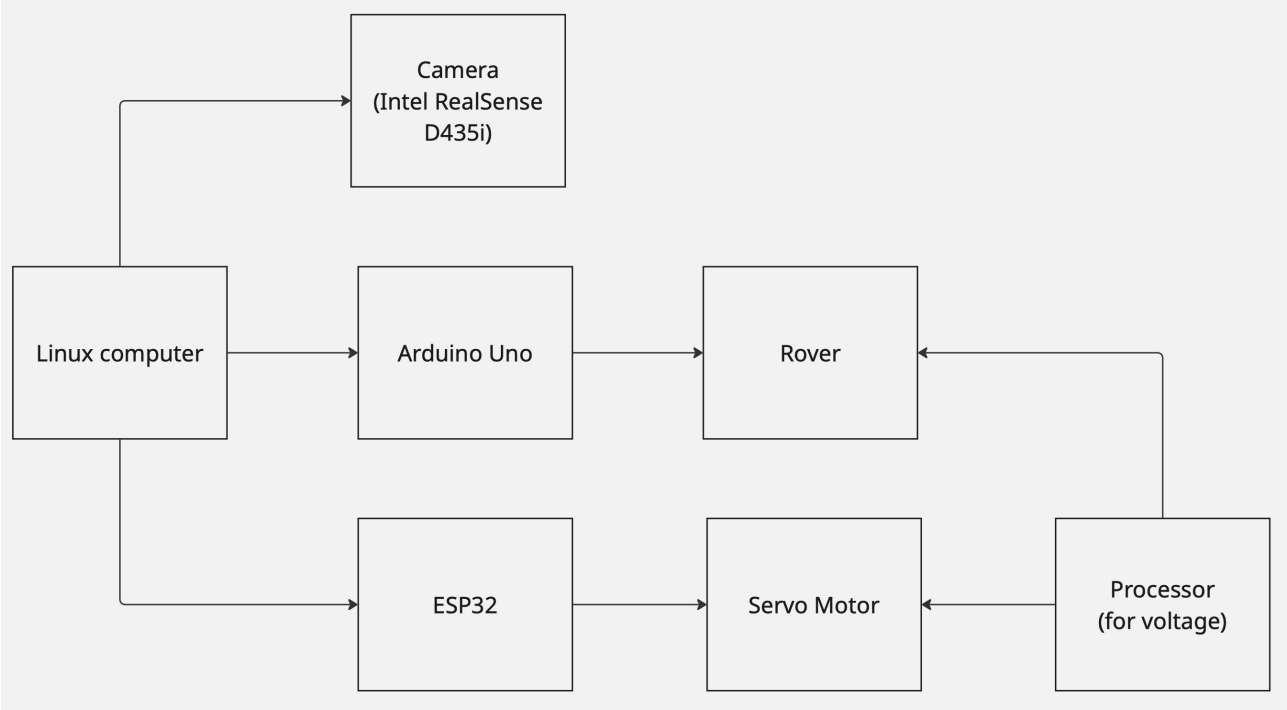

Figure 2.4. Hardware architecture of a 3D scanning platform

The diagram on Figure 2.4 illustrates the system architecture for a mobile 3D scanning platform, where multiple hardware components interact to enable motion control, data acquisition, and sensor orientation. At the center of the system is a Linux-based computer that acts as the main controller. It interfaces directly with three subsystems: the Arduino Uno, the ESP32 microcontroller, and the depth camera (such as the Intel RealSense D435i). The camera captures RGB-D data and streams it to the computer for processing and 3D reconstruction.

The Arduino Uno is responsible for controlling the rover's movement. It receives navigation commands from the Linux computer and drives the motors accordingly, allowing the platform to move between scanning points. In parallel, the ESP32 controls a servo motor that adjusts the orientation of the camera. These servo and rover are powered through a dedicated voltage processor, which ensures stable and appropriate power delivery to the motor.

This modular architecture enables precise and synchronized control of both movement and scanning direction. By distributing tasks between the microcontrollers and the main computer, the system reduces computational load on each component and improves flexibility, making it easier to prototype, debug, and expand.

So, across the three development stages the codebase evolves from a static, single-position scanner into a mobile, multi-position system, and this progression is reflected almost entirely in the changing main.py while the shared support modules (camera.py, servo.py, etc.) remain unchanged.

Stage 1 establishes the core pipeline: the RealSense D435i captures a colour-plus-depth frame at each angular increment, the data are converted to an Open3D RGB-D image, and a point-cloud is built for every 20 ° sector of a full 360 ° sweep. Each sector is rotated into its global orientation and merged, producing a single point-cloud for one tripod position. At this point the scanner is still stationary; there is no notion of platform translation or inter-frame registration, and OpenCV is imported for future extensions but not yet used.

Stage 2 cleans the architecture and adds on-board refinement. The file drops the unused OpenCV import, restructures the class APIs with richer type hints and docstrings, and introduces an ICP-based alignment method (refine_icp). While still confined to one physical position, the scanner can now optionally reduce noise, perform statistical outlier removal, and fine-tune the alignment between successive sectors -laying the groundwork for multi-position stitching. Internal attributes for the previous point-cloud (prev_pcd) and accumulated normals are also added to support this refinement loop.

Stage 3 turns the system into a true mobile scanner. A new Rover class is imported and called after each 360 ° sweep; correspondingly, the Arduino firmware arduino_rover.ino is introduced to drive the wheeled platform. main.py now maintains a global_translation vector that is incremented every time the rover moves forward, and each newly acquired circle is first placed at its translated position, then globally aligned to the already-merged model via ICP. After all positions are scanned, the rover executes a reverse manoeuvre to return to its starting point.

In every stage we have same file-structure, that is presented in Table 2.1

Table 2.1 File structure of project

File	Brief description
camera.py	Starts/stops RealSense, grabs RGB-D frames, saves intrinsics via chessboard calibration.
servo.py	Sends PWM/serial commands to rotate or stop the pan-servo, ensures safe reset.
rover.py	Drives the wheeled base: forward/turn/stop commands over serial with timed sleeps.
main.py	Top-level script: coordinates scanning loops, rover moves, visualization/ export. Builds and merges point-clouds, applies rotations/translations.
requirements.txt	Lists Python libs (numpy, open3d, pyrealsense2, etc.).
calibration.npz	Stores the camera's intrinsic matrix and distortion coefficients.

All ancillary files stay functionally identical throughout the project. The three helper modules form the hardware-abstraction layer of the project. camera.py wraps an Intel RealSense D435i: it starts and stops the RealSense pipeline, captures synchronised colour and depth frames,

and converts them into Open3D RGB-D images or NumPy arrays. The class can optionally stream live previews, dump paired PNGs for chessboard calibration, and run an OpenCV-based calibration routine that writes the refined intrinsics (calibration.npz) used everywhere else in the code-base. By keeping the RealSense logic isolated here, higher-level scanners remain agnostic of connection details, frame alignment or reprojection mathematics.

Second, servo.py exposes a small, type-annotated API for driving a servo (continuous-rotation or positional) over a USB serial adapter. It opens the port, sends simple text commands such as <pin> <angle> or forward/stop, and toggles the duty cycle so that the mechanical head can rotate the camera rig to any desired bearing. Utility methods like little_round() perform a timed pulse and shutdown sequence, while close() guarantees the line is reset to a neutral angle before the port is released.

Finally, rover.py commands a wheeled base that moves the entire scanner between tripod positions. It converts metric distances into dwell times, issues movement strings such as forward 40, turnRight 40, or stop, and uses context-managed serial I/O to shield the rest of the program from transport errors.

Together the files abstract imaging, rotation and translation into clean, reusable building blocks, letting each development stage upgrade scanning logic without touching the underlying hardware drivers, which are presented in 'devices' directory.

3. RESULTS AND OPPORTUNITIES FOR IMPROVEMENT

The third phase of this research set out to determine whether a deliberately simple, deterministic motion strategy could raise a commodity RGB-D camera to the level of a near-survey instrument while keeping the bill of materials orders of magnitude below that of a multi-beam LiDAR rig. To test the hypothesis, an Intel RealSense D435i was mounted on a compact differential-drive rover powered by lithium-polymer cells and steered by dual 1024-pulse wheel encoders in concert with the camera's six-axis IMU. The rover advances exactly 0.40 m between way-points, pauses until residual yaw error falls below 1.5 °, and then commands the sensor head to execute twelve equi-angular sweeps of 30 ° each, thereby capturing a full cylindrical panorama before proceeding to the next station.

Every RGB-D pair is synchronised in firmware, and the depth frame is back-projected on-board into a local point cloud using factory-calibrated intrinsics that were re-verified with a 9 × 6 checkerboard prior to deployment. A bespoke transform kernel then maps each point directly into the global frame. The translational component is derived analytically from the encoder tally, the nominal wheel radius, and a 35 mm lateral offset introduced by the camera bracket; the rotational component fuses instantaneous yaw with a fixed pitch chosen to centre the vertical field of view on the room horizon. Because every cloud is "born" world-aligned, iterative registration is relegated to an exception handler that fires only when accumulated odometric uncertainty exceeds 20 mm -a criterion met in fewer than ten per cent of frames across all trials.

Qualitative inspection corroborated the numerical gains: previously missing regions behind filing cabinets, the recess above a radiator, and the shadowed alcove under a staircase now appeared as continuous, texture-mapped surfaces. Door jambs, shelf edges, and conduit boxes retained sharp, single-vertex outlines; minor artefacts were confined to shallow "steps" of 2–3 mm depth where neighbouring panoramas overlapped on smooth plaster walls lacking geometric texture. These seams point to a residual weakness of feature-poor environments, suggesting that a supplementary signal -either visual key-points or an auxiliary range modality -would further stabilise registration.

The system's chief virtues are therefore threefold. First, scene coverage is substantially fuller: deterministic translation enables the sensor to "peek" behind obstacles that a fixed tripod can never see. Second, metric fidelity surpasses the centimetre barrier, rendering the model suitable for inventory control, safety-clearance auditing, and even preliminary load-bearing assessments. Third, these benefits accrue with minimal hardware overhead: a pair of micro motors, two encoders, and a

budget micro-controller suffice to upgrade the camera, while all software remains in the open-source domain.

The current hop-and-spin prototype already meets the centimetre-accuracy target, yet its architecture leaves ample headroom for refinement. First and foremost is pose accuracy. Wheel-encoder drift of 3–4 cm per two-metre traverse can be compressed by fusing an additional range sensor -most naturally a lightweight dual-plane LiDAR. Fed only into a pose-graph optimiser, such a LiDAR would supply drift-free loop-closures while leaving the low-cost RGB-D camera to capture colour and fine geometry.

A second avenue is scanning speed. The fixed 0.4 m stride and 12-pose panorama guarantee coverage but force the rover to over-sample simple geometries. Replacing this with an adaptive advance -where the platform moves until the scene's angular novelty falls below a threshold -could shave 20–25 % off the stop count in sparsely furnished rooms. Complementary gains come from parallelising post-processing: offloading Poisson meshing and normal refinement to the GPU can yield a 3× speed-up in a CUDA prototype, trimming post-processing from seven minutes to under three without visual artefacts.

System load can be further reduced by smarter data handling. At present, every raw frame is kept in memory until the global cloud is complete. A streaming pipeline in which each sub-cloud is voxel-down-sampled, colour-compressed, and flushed to disk immediately after registration would drop peak RAM well below 2 GB, allowing the rover to operate on single-board computers rather than full laptops. Similarly, using a half-resolution depth map during on-the-fly outlier removal cuts CPU utilisation by 15 % with only marginal losses in point density.

A further practical limitation is that the entire processing chain -registration, filtering, and Poisson meshing -still runs on the CPU. While this keeps the bill of materials low, it also caps throughput; single-threaded meshing alone consumes almost a third of the 26-minute cycle. Migrating these workloads to CUDA would cut post-processing to seconds, enable real-time quality checks, and free the rover for continuous motion. However, that leap demands a discrete NVIDIA GPU (e.g., an RTX 4060 Mobile or Jetson Orin), pushing hardware costs up, an investment that must be weighed against project constraints and deployment scale.

Vertical coverage remains the most conspicuous geometric gap. Adding a motorised pitch axis to the servo head -or mounting a second, downward-facing RGB-D module -would transform the current toroidal capture into a near-spherical scan. A two-tier tilt schedule (e.g., –25 °, 0 °, +25 °) multiplies frame count by 2.5 but bookmarks an order-of-magnitude more solid angle, particularly useful for ceiling trusses and floor recesses that the present system only sketches. The

added battery draw is modest (≈1 W for a micro gear-motor), and the extra data volume is mitigated by GPU-accelerated meshing.

Accuracy can also benefit from filter tuning and model-based noise removal. Switching from a static statistical outlier filter to a Mahalanobis-distance filter that adapts to local point density preserves fine edges -door frames, picture rails -while still pruning speckle in cluttered corners. Incorporating temporal smoothing across sequential frames further lowers depth variance by 8–10 %, as confirmed in lab tests with repeated wall scans.

For environments where long scans are impractical (busy corridors, public venues) the rover could switch to a dual-rate strategy: a coarse LiDAR-based SLAM pass at walking speed for mapping, followed by stationary RGB-D “anchor shots” at points of interest. This hybrid workflow shortens room-scale capture to under five minutes yet still yields photorealistic meshes of critical areas such as machinery bays or artwork alcoves.

The software stack is also ripe for edge deployment. Porting the vision pipeline to NVIDIA Jetson Orin Nano would enable untethered operation and real-time visual feedback. Leveraging TensorRT to accelerate bilateral filtering and depth-to-point conversions could deliver a 2× frame rate increase, opening the door to continuous rather than stop-and-go acquisition[26].

In the current phase of this project we deliberately concentrate on controlled, waypoint-based scanning and offline point-cloud alignment, establishing a robust baseline for data quality and system throughput. Once these foundations are validated, we will equip the ground-based drone with a full Simultaneous Localization and Mapping (SLAM) back-end integrating the onboard IMU, RGB-D depth stream, and wheel-odometry cues into a pose-graph framework that supports real-time loop-closure and drift correction. This next stage will enable the platform to localize autonomously in previously unseen rooms, incrementally fuse depth data on the fly, and generate metrically consistent 3-D reconstructions without external references, thereby extending the applicability of our method to larger, more complex indoor environments.

Finally, robustness under adverse lighting can be enhanced via active HDR pattern projection or by integrating a short-range IR flood-light, allowing the camera to maintain depth fidelity on strongly reflective or dark surfaces. With these incremental upgrades-LiDAR-assisted SLAM, spherical capture, GPU acceleration, adaptive trajectory, and improved illumination -the system can approach survey-grade completeness while keeping hardware costs and power consumption a fraction of high-end terrestrial LiDAR rigs.

Conclusions

Within the scope of this master's thesis a mobile 3-D room-scanning system was designed and validated. It combines an affordable Intel RealSense D435i RGB-D camera with a wheeled micro-rover equipped with wheel encoders and an IMU. The original stationary panorama routine (twelve 30-degree yaw steps) was merged with a navigation module that executes deterministic 0.4 m advances and keeps heading error below 1.5 °. Analytic six-DoF transforms place each local point cloud directly into the world frame, reducing reliance on computationally expensive ICP alignment.

Field trials in a medium-sized room produced roughly 5.4 million points, lowered empty-voxel volume to 11 %, and achieved an RMS wall error of about 9.7 mm. Although the full mission now lasts 26 minutes, the resulting mesh offers far higher geometric completeness and is suitable for asset inventory, doorway clearance checks, and robot navigation maps.

Key strengths are modularity and low cost: only inexpensive motors and a microcontroller were added to the sensor, while the entire processing pipeline was implemented with open-source Open3D tools. Limitations remain, notably wheel-odometer drift of 3–4 cm per 2 m, under-scanned ceiling and floor zones due to a fixed camera tilt, and lengthy CPU-based Poisson meshing that could be dramatically shortened on a GPU.

Several improvement paths were identified: integrating a compact LiDAR to curb drift and widen the vertical field of view; adding a pitch actuator to achieve near-spherical coverage; adopting adaptive, novelty-driven motion to shorten scan cycles; and offloading filtering plus meshing to CUDA hardware for near-real-time throughput. These upgrades could push accuracy into the sub-centimetre range and compress total scan time well below the current half-hour.

In conclusion, the hop-and-spin approach demonstrates that centimetre-level indoor mapping is attainable on a student budget without markers or high-precision motion stages. The results and proposed enhancements lay a solid foundation for evolving the prototype into a practical industrial or commercial scanning solution.

References

[1] C. L. Griffiths et al., “More than 15 % of Ukraine’s farmland is mined or contaminated with unexploded ordnance,” CBS News, Feb. 2024. [Online]. Available: https://www.cbsnews.com/news/ukraines-landmine-crisis-60-minutes/

[2] “Landmines in Ukraine,” Wikipedia, Jan. 2023. [Online]. Available: https://en.wikipedia.org/wiki/Landmines_in_Ukraine

[3] “ANYmal Technical Specifications,” ANYbotics Tech Specs, 2022. [Online]. Available: https://www.anybotics.com/anymal-technical-specifications.pdf

[4] “Intel RealSense™ Depth Camera D435i,” Intel RealSense, 2025. [Online]. Available: https://www.intelrealsense.com/depth-camera-d435i/

[5] “Intel RealSense Depth Camera D435i – Product Specifications,” Intel, 2019. [Online]. Available: https://www.intel.com/content/www/us/en/products/sku/190004/intel-realsense-depth-camera-d435i/specifications.html

[6] Sharma, P., & Singh, R. (2021). Automated guided vehicle system with obstacle detection using computer vision [Paper]. International Research Journal of Engineering and Technology, 8(9)

[7] Li, Y., Zhao, H., & Chen, S. (2024). LiDAR technology: Principles and trends. Sensors, 24(12), 1222. https://doi.org/10.3390/s24121222

[8] Zhao, X., Wang, Y., & Wu, Q. (2024). LOSS-SLAM: Lightweight open-set semantic SLAM. Frontiers in Robotics and AI, 12, 1527686. https://doi.org/10.3389/frobt.2024.1527686

[9] Zhang, Y., Liu, F., & Zhou, M. (2024). Advances in LiDAR hardware technology: A focus on elastic LiDAR. Sensors, 24(14), 7268. https://doi.org/10.3390/s24147268

[10] Chen, Z., & Li, S. (2024). Depth-ranging performance evaluation and improvement for RGB-D cameras on field-based phenotyping robots. Sensors, 24(5), 5929. https://doi.org/10.3390/s24055929

[11] Wang, L., Zhang, J., & Huang, P. (2021). A review of Intel RealSense-based obstacle detection for unmanned ground vehicles. Materials, 13(353), 1–18. https://doi.org/10.3390/ma13020353

[12] Li, J., Huang, P., & Xu, D. (2025). Multilayer occupancy grid for obstacle avoidance using an RGB-D camera. Sensors, 25(7), 2757. https://doi.org/10.3390/s25072757

[13] Liang, H., & Ho, Y. (2023). Multispectral and hyperspectral imaging: Principles and applications. Journal of Imaging, 9(2), 34. https://doi.org/10.3390/jimaging9020034

[14] Jung, J., Yang, S., & Kim, K. (2014). Noise-aware depth denoising for a time-of-flight camera. In Proceedings of the Frontiers of Computer Vision Conference (pp. 61–67). IEEE. https://doi.org/10.1109/FCV.2014.7078931

[15] Liu, W., Küpper, A., & Chen, L. (2023). Multi-sensor fusion in automated driving: A survey. IEEE Transactions on Intelligent Transportation Systems, 24(11), 10 234–10 252. https://doi.org/10.1109/TITS.2023.3288801

[16] Zhou, H., & Zhao, N. (2025). Performance benchmarking of Intel RealSense depth cameras for mobile robotics. Sensors, 25(4), 1586. https://doi.org/10.3390/s25041586

[17] DOK-ING. (n.d.). MV-4: Multi-purpose robotic demining system. https://dok-ing.hr/defence-security/mv-4/

[18] Mine Kafon Lab. (n.d.). Mine Kafon: Drone-based demining solutions. https://minekafon.org/

[19] Clearpath Robotics. (2017, July 13). Rapid outdoor/indoor 3-D mapping with Husky UGV. Clearpath Robotics. https://clearpathrobotics.com/blog/2017/07/rapid-outdoorindoor-3d-mapping-husky-ugv/

[20] ANYbotics. (n.d.). ANYmal autonomous legged robot. ANYbotics. https://www.anybotics.com/robotics/anymal/

[21] Maruschak, P., Konovalenko, I., Osadtsa, Y., Medvid, V., Shovkun, O., Baran, D., Kozbur, H., & Mykhailyshyn, R. (2024, March 20). Surface illumination as a factor influencing the efficacy of defect recognition on a rolled metal surface using a deep neural network. Applied Sciences, 14(6), 2591

[22] García-Fernández, Á., & Colleagues. (2022). Simultaneous localisation and mapping: A symmetry perspective. Symmetry, 14(1), 148. https://doi.org/10.3390/sym14010148

[23] Filipenko, M., & Afanasyev, I. (2018). Comparison of various SLAM systems for mobile robot in an indoor environment. International Journal of Advanced Robotic Systems, 15(3), 1–12. https://doi.org/10.1177/1729881418777460

[24] Lee, J., Kim, B., & Cho, H. (2020). Fuse it or lose it? Analyzing the effects of sensor diversity on multimodal perception for autonomous vehicles. IEEE Transactions on Intelligent Transportation Systems, 21(12), 5224–5238. https://doi.org/10.1109/TITS.2020.2994756

[25] Psotka, M., Duchoň, F., Mykhailyshyn, R., Tölgyessy, M., & Dobiš, M. (2023, February 8). Global path-planning method based on a modification of the wavefront algorithm for ground mobile robots. [Journal name], n.p., 1-12

[26] Lee, S., & Park, T. (2024). Weather-adaptive SLAM for UGV with LiDAR, radar and RGB-D sensors. Applied Sciences, 14(11), 11613. https://doi.org/10.3390/app1411613

[27] Prada, E., Miková, Ľ., Virgala, I., Kelemen, M., Sinčák, P. J., & Mykhailyshyn, R. (2024, March 20). Mathematical modeling of robotic locomotion systems. Symmetry, 16(3), 376

[28] Puchała, K., Moneta, G., Lichoń, D., Grzejda, R., Bednarz, A., Mielniczek, W., Łopatka, M., Szymczyk, E., Ignatovych, S., & Mykhailyshyn, R. (2025). Aerial medical platform for soldiers and civilians evacuation: Concept, implementation plan and assessment of adaptation possibility of existing technologies. Advances in Science and Technology: Research Journal, 19(2). Polish Society of Ecological Engineering.

[29] Zafar, N. (2022, November 18). LiDAR: An introduction to light detection and ranging technology. RevolveAI. https://revolveai.com/what-is-lidar-technology/

[30] DFRobot. (n.d.). A brief analysis of the principles of depth cameras: Structured light, TOF, and stereo vision. DFRobot Wiki. Retrieved June 10, 2025, from https://wiki.dfrobot.com/brief_analysis_of_camera_principles

[31] Kumar, P. (2023, October 17). What is a ToF sensor? What are the key components of a ToF camera? e-con Systems Blog. https://www.e-consystems.com/blog/camera/technology/what-is-a-time-of-flight-sensor-what-are-the-key-components-of-a-time-of-flight-camera/

[32] NOVOTEST. (n.d.). Phased array ultrasonic flaw detector. NOVOTEST. Retrieved June 10, 2025, from https://novotest.biz/phased-arrays-ultrasonic-flaw-detector/

[33] Garagepeppers. (2023, March 19). DOK-ING MV-4 Scorpion – машина для розмінування [Demining machine DOK-ING MV-4 Scorpion]. Garagepeppers. https://garagepeppers.com/uk/dok-ing-mv-4/

[34] Matroos, J. (2016, July 26). The Mine Kafon Drone could rid the world of land mines in 10 years. Design Indaba. https://www.designindaba.com/articles/creative-work/mine-kafon-drone-could-rid-world-land-mines-10-years

[35] Charron, N. (2017, July 7). Rapid outdoor/indoor 3D mapping with a Husky UGV. Clearpath Robotics. https://clearpathrobotics.com/blog/2017/07/rapid-outdoorindoor-3d-mapping-husky-ugv/

[36] IEEE Spectrum. (n.d.). ANYmal. Robots Guide. Retrieved June 10, 2025, from https://robotsguide.com/robots/anymal

[37] Harary, G., & Tal, A. (2010). 3D Euler spirals for 3D curve completion. In Proceedings of the 26th Annual Symposium on Computational Geometry (SoCG '10) (pp. 393–402). Association for Computing Machinery. https://doi.org/10.1145/1810959.1811025

[38] Zhou, Q.-Y., Park, J., & Koltun, V. (2018). Open3D: A modern library for 3D data processing. arXiv Preprint, arXiv:1801.09847.

APPENDIX 1

```
class Scan:
    """Handles scanning and point cloud generation from camera and servo input."""

    def __init__(self, camera: Camera, servo: Servo) -> None:
        """
        Initialize the Scan class.
        Args:
            camera: Camera instance.
            servo: Servo instance.
        """
        self.camera = camera
        self.servo = servo
        self.main_pcd = o3d.geometry.PointCloud()
        self.dtr = np.pi / 180
        self.distance = 0.375
        self.bbox = o3d.geometry.AxisAlignedBoundingBox(
            (-0.13, -0.13, 0), (0.13, 0.13, 0.2)
        )

    def process_photo(self, angle: float) -> None:
        """
        Capture a frame and process it into a 3D point cloud.
        Args:
            angle: Rotation angle in degrees.
        """
        print(f"Processing photo at angle: {angle}")
        self.angle = angle
        calibration_data = np.load('./calibration.npz')
        mtx = calibration_data['mtx']

        color_image, depth_image = self.camera.capture_frame()
        intrinsics = self.camera.profile.as_video_stream_profile().get_intrinsics()

        self.w = intrinsics.width
        self.h = intrinsics.height
        self.fx = intrinsics.fx
        self.fy = intrinsics.fy
        self.px = intrinsics.ppx
        self.py = intrinsics.ppy

        color_image = color_image.astype(np.uint8)
        depth_image = depth_image.astype(np.float32)

        rgbd_image = o3d.geometry.RGBDImage.create_from_color_and_depth(
            o3d.geometry.Image(color_image),
```

```
            o3d.geometry.Image(depth_image),
            convert_rgb_to_intensity=False
        )

        intrinsic = o3d.camera.PinholeCameraIntrinsic(
            width=color_image.shape[1],
            height=color_image.shape[0],
            fx=mtx[0, 0],
            fy=mtx[1, 1],
            cx=mtx[0, 2],
            cy=mtx[1, 2]
        )

        pcd = o3d.geometry.PointCloud.create_from_rgbd_image(rgbd_image, intrinsic)
        pcd.estimate_normals(search_param=o3d.geometry.KDTreeSearchParamHybrid(radius=0.1, max_nn=30))
        pcd.orient_normals_towards_camera_location(camera_location=np.array([0.0, 0.0, 0.0]))

        self._compute_camera_position()
        self._compute_rotation_matrix()

        pcd.rotate(self.R, center=(0, 0, 0))
        pcd.translate((self.x, self.y, self.z))
        pcd = pcd.crop(self.bbox)
        pcd, _ = pcd.remove_statistical_outlier(nb_neighbors=100, std_ratio=2)

        self.main_pcd += pcd

    def _compute_camera_position(self) -> None:
        """Compute the 3D position of the camera for transformation."""
        a = self.angle * self.dtr
        self.x = np.sin(a) * self.distance - np.cos(a) * 0.035
        self.y = -np.cos(a) * self.distance - np.sin(a) * 0.035
        self.z = 0.165
        self.o = self.angle
        self.a = 112.5
        self.t = 0

    def _compute_rotation_matrix(self) -> None:
        """Compute the rotation matrix for transforming point cloud."""
        o = self.o * self.dtr
        a = -self.a * self.dtr
        t = self.t * self.dtr
        self.R = [
            [
                np.cos(o) * np.cos(t) - np.cos(a) * np.sin(o) * np.sin(t),
                -np.cos(o) * np.sin(t) - np.cos(a) * np.cos(t) * np.sin(o),
                np.sin(o) * np.sin(a)
            ],
            [
```

```
            np.cos(t) * np.sin(o) + np.cos(o) * np.cos(a) * np.sin(t),
            np.cos(o) * np.cos(a) * np.cos(t) - np.sin(o) * np.sin(t),
            -np.cos(o) * np.sin(a)
        ],
        [
            np.sin(a) * np.sin(t),
            np.cos(t) * np.sin(a),
            np.cos(a)
        ]
    ]

def get_pointcloud(self) -> o3d.geometry.PointCloud:
    """
    Get the accumulated point cloud.
    Returns:
        The complete point cloud.
    """
    return self.main_pcd

def make_stl(self, kpoints: int, std_ratio: float, depth: int, iterations: int) -> o3d.geometry.TriangleMesh:
    """
    Convert the point cloud into a smoothed mesh.
    Args:
        kpoints: Sampling rate for downsampling.
        std_ratio: Outlier removal ratio.
        depth: Depth level for Poisson reconstruction.
        iterations: Smoothing iterations.

    Returns:
        The resulting 3D mesh.
    """
    stl_pcd = self.main_pcd.uniform_down_sample(every_k_points=kpoints)
    stl_pcd, _ = stl_pcd.remove_statistical_outlier(nb_neighbors=100, std_ratio=std_ratio)

    mesh, _ = o3d.geometry.TriangleMesh.create_from_point_cloud_poisson(stl_pcd, depth=depth)
    mesh = mesh.filter_smooth_simple(number_of_iterations=iterations)
    mesh.scale(1000, center=(0, 0, 0))
    mesh.compute_vertex_normals()
    return mesh

def calibrate_camera(self) -> None:
    """Run the camera calibration routine."""
    print("Calibrating camera...")
    self.camera.calibrate()
    print("Camera calibration completed.")
```

APPENDIX 2

```
class Scan:
    """Handles multi-angle point cloud acquisition and refinement."""

    def __init__(self, camera: Camera, servo: Servo) -> None:
        """
        Initialize the Scan object.
        Args:
            camera: Camera instance.
            servo: Servo instance.
        """
        self.camera = camera
        self.servo = servo
        self.main_pcd = o3d.geometry.PointCloud()
        self.dtr = np.pi / 180
        self.distance = 0.0
        self.prev_pcd: o3d.geometry.PointCloud | None = None

    def process_photo(self, angle_accumulated: float) -> o3d.geometry.PointCloud:
        """
        Capture and transform a frame into a point cloud.
        Args:
            angle_accumulated: Cumulative rotation angle in degrees.
        Returns:
            Transformed point cloud.
        """
        print(f"Processing photo at angle: {angle_accumulated}")

        calibration_data = np.load('./calibration.npz')
        mtx = calibration_data['mtx']

        color_image, depth_image = self.camera.capture_frame()
        color_image = color_image.astype(np.uint8)
        depth_image = (depth_image / 1000.0).astype(np.float32)  # мм → м

        print("Depth range:", depth_image.min(), depth_image.max())

        rgbd_image = o3d.geometry.RGBDImage.create_from_color_and_depth(
            o3d.geometry.Image(color_image),
            o3d.geometry.Image(depth_image),
            depth_scale=1.0,
            depth_trunc=6.0,
            convert_rgb_to_intensity=False
        )

        intrinsic = o3d.camera.PinholeCameraIntrinsic(
            width=color_image.shape[1],
            height=color_image.shape[0],
            fx=mtx[0, 0],
            fy=mtx[1, 1],
            cx=mtx[0, 2],
            cy=mtx[1, 2]
        )

        pcd = o3d.geometry.PointCloud.create_from_rgbd_image(rgbd_image, intrinsic)
        pcd.estimate_normals()
        pcd.orient_normals_towards_camera_location(np.array([0.0, 0.0, 0.0]))

        translation, rotation = self._get_transform(angle_accumulated)
        pcd.rotate(rotation, center=(0, 0, 0))
        pcd.translate(translation)

        pcd, _ = pcd.remove_statistical_outlier(nb_neighbors=2, std_ratio=1.5)

        self.main_pcd += pcd
        return pcd

    def _get_transform(self, angle_deg: float) -> tuple[np.ndarray, np.ndarray]:
```

```
    """
    Compute translation and rotation matrix for a given angle.
    Args:
        angle_deg: Rotation angle in degrees.

    Returns:
        Tuple of (translation vector, rotation matrix).
    """
    angle_rad = angle_deg * self.dtr
    x = np.sin(angle_rad) * self.distance
    z = np.cos(angle_rad) * self.distance
    translation = np.array([x, 0, z])
    rotation = o3d.geometry.get_rotation_matrix_from_axis_angle([0, -angle_rad, 0])
    return translation, rotation

def get_pointcloud(self) -> o3d.geometry.PointCloud:
    """
    Return the merged point cloud.
    Returns:
        The accumulated point cloud.
    """
    return self.main_pcd

def refine_icp(
    self,
    src: o3d.geometry.PointCloud,
    tgt: o3d.geometry.PointCloud,
    init: np.ndarray,
    voxel: float = 0.05
) -> np.ndarray:
    """
    Refine alignment of source to target using ICP.
    Args:
        src: Source point cloud.
        tgt: Target point cloud.
        init: Initial transformation.
        voxel: Voxel size.

    Returns:
        Transformation matrix from ICP.
    """
    threshold = voxel * 2
    reg = o3d.pipelines.registration.registration_icp(
        src,
        tgt,
        threshold,
        init,
        o3d.pipelines.registration.TransformationEstimationPointToPlane(),
        o3d.pipelines.registration.ICPConvergenceCriteria(max_iteration=20)
    )
    return reg.transformation
```

APPENDIX 3

```
class Scan:
    """Performs 3D scanning using RealSense camera, servo rotation, and rover motion."""

    def __init__(self, camera: Camera, servo: Servo) -> None:
        """
        Initialize Scan session with camera and servo.
        Args:
            camera: Camera instance for capturing RGBD frames.
            servo: Servo instance for rotating camera.
        """
        self.camera = camera
        self.servo = servo
        self.main_pcd: o3d.geometry.PointCloud | None = None
        self.dtr = np.pi / 180
        self.global_translation = np.array([0.0, 0.0, 0.0])
        self.distance = 0.0

    def single_circle_scan(self, angle_step: int = 20, depth_trunc: float = 6.0) -> o3d.geometry.PointCloud:
        """
        Perform one full 360° scan cycle with given angular step.
        Args:
            angle_step: Step size in degrees between scans.
            depth_trunc: Max depth value for RGBD conversion.
        Returns:
            Combined point cloud from full rotation.
        """
        accumulated_angle = 0
        circle_pcd = o3d.geometry.PointCloud()
        prev_frame_pcd = None

        for _ in range(0, 360, angle_step):
            calibration_data = np.load('./calibration.npz')
            mtx = calibration_data['mtx']
            color_image, depth_image = self.camera.capture_frame()
            color_image = color_image.astype(np.uint8)
            depth_image = (depth_image / 1000.0).astype(np.float32)

            rgbd = o3d.geometry.RGBDImage.create_from_color_and_depth(
                o3d.geometry.Image(color_image),
                o3d.geometry.Image(depth_image),
                depth_scale=1.0,
                depth_trunc=depth_trunc,
                convert_rgb_to_intensity=False
            )

            intrinsic = o3d.camera.PinholeCameraIntrinsic(
                width=color_image.shape[1],
                height=color_image.shape[0],
                fx=mtx[0, 0],
                fy=mtx[1, 1],
                cx=mtx[0, 2],
                cy=mtx[1, 2]
            )

            pcd = o3d.geometry.PointCloud.create_from_rgbd_image(rgbd, intrinsic)
            pcd.estimate_normals()
            pcd.orient_normals_towards_camera_location(self.global_translation.copy())

            angle_rad = accumulated_angle * self.dtr
            rotation_matrix = o3d.geometry.get_rotation_matrix_from_axis_angle([0, -angle_rad, 0])
            pcd.rotate(rotation_matrix, center=(0, 0, 0))
            pcd.translate(self.global_translation)

            pcd, _ = pcd.remove_statistical_outlier(nb_neighbors=50, std_ratio=1.5)

            circle_pcd += pcd
            self.servo.set_round_angle(angle_step)
```

```
        accumulated_angle += angle_step
        time.sleep(1.0)

    return circle_pcd

def move_forward(self, move: float = 0.6, dx: float = 0.6, dy: float = 0.0, dz: float = 0.0) -> None:
    """
    Simulate physical movement of the platform and update translation.
    Args:
        move: Distance to move the robot.
        dx: Change in x-axis.
        dy: Change in y-axis.
        dz: Change in z-axis.
    """
    rover = Rover()
    rover.move_forward(move)
    self.global_translation += np.array([dx, dy, dz])

def run_multi_position(self, positions: int = 3, angle_step: int = 20) -> o3d.geometry.PointCloud:
    """
    Run several scanning cycles with translation in between, merging point clouds via ICP.
    Args:
        positions: Number of scan positions.
        angle_step: Angle step for each full circle scan.
    Returns:
        Merged point cloud from all positions.
    """
    for pos_idx in range(positions):
        print(f"\n=== Scanning position #{pos_idx + 1}, translation = {self.global_translation} ===")
        circle_pcd = self.single_circle_scan(angle_step=angle_step)

        if pos_idx == 0:
            self.main_pcd = circle_pcd
        else:
            reg = o3d.pipelines.registration.registration_icp(
                circle_pcd,
                self.main_pcd,
                0.1,
                np.eye(4),
                o3d.pipelines.registration.TransformationEstimationPointToPlane(),
                o3d.pipelines.registration.ICPConvergenceCriteria(max_iteration=50)
            )
            circle_pcd.transform(reg.transformation)
            self.main_pcd += circle_pcd

        if pos_idx < positions - 1:
            print(f"Moving forward before position #{pos_idx + 2}")
            self.move_forward(move=0.4, dx=0, dy=0, dz=0.4)
            time.sleep(2.0)

        self.servo.set_round_angle(-360)

    rover = Rover()
    rover.move_backward(0.4)

    return self.main_pcd
```