VoxDepth: Rectification of Depth Images on Edge Devices

YASHASHWEE CHAKRABARTY, Computer Science and Engineering, Indian Institute of Technology, India
AKANKSHA DIXIT, Electrical Engineering, Indian Institute of Technology, India
SMRUTI R. SARANGI, Computer Science and Engineering (Joint appointment with Electrical Engineering), Indian Institute of Technology Delhi, India

Autonomous mobile robots like self-flying drones and industrial robots heavily depend on depth images to perform tasks such as 3D reconstruction and visual SLAM. However, the presence of inaccuracies in these depth images can greatly hinder the effectiveness of these applications, resulting in sub-optimal results. Depth images produced by commercially available cameras frequently exhibit noise, which manifests as flickering pixels and erroneous patches. Machine Learning (ML)-based methods to rectify these images are unsuitable for edge devices that have very limited computational resources. Non-ML methods are much faster but have limited accuracy, especially for correcting errors that are a result of occlusion and camera movement. We propose a scheme called *VoxDepth* that is fast, accurate, and runs very well on edge devices such as the NVIDIA Jetson Nano board. It relies on a host of novel techniques: 3D point cloud construction and fusion, and using it to create a 2D template to fix erroneous depth images. *VoxDepth* shows superior results on both synthetic and real-world datasets. We specifically demonstrate a 31% improvement in quality as compared to state-of-the-art methods on real-world depth datasets, while maintaining a competitive framerate of 27 FPS (frames per second).

CCS Concepts: • **Computer systems organization → Embedded hardware**; **Real-time system specification**.

Additional Key Words and Phrases: Point cloud, Image registration, Stereo camera

ACM Reference Format:
Yashashwee Chakrabarty, Akanksha Dixit, and Smruti R. Sarangi. 2025. *VoxDepth*: Rectification of Depth Images on Edge Devices. 1, 1 (September 2025), 28 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

The[1] estimated market size of autonomous mobile robots is USD 3.88 billion in 2024. It is projected to reach USD 8.02 billion by 2029. This growth is expected to occur at a compound annual growth rate (CAGR) of 15.60% throughout this period [34]. For such autonomous systems, high-quality depth images are crucial – they allow the robots to create an accurate 3D map of the environment. They are specifically needed to support different robotics applications such as RGB-D SLAM [15] (simultaneous localization and mapping), 3D object detection [40], drone swarming [8], 3D environment mapping and surveillance. Depth images are produced by depth cameras, which are specialized cameras

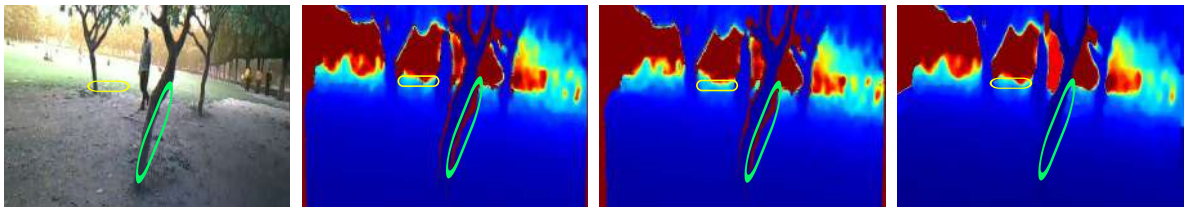
[1]Kindly take a color printout or use a color monitor to view the PDF file. A monochrome printout will not be able to show the key features of images and our contributions. No subset of this paper has been published in any peer-reviewed conference/journal. This is a fresh submission.

Authors' addresses: Yashashwee Chakrabarty, Computer Science and Engineering, Indian Institute of Technology, New Delhi, India, yashashwee99@gmail.com; Akanksha Dixit, Electrical Engineering, Indian Institute of Technology, New Delhi, India, Akanksha.Dixit@ee.iitd.ac.in; Smruti R. Sarangi, Hi-Tech Robotics and Autonomous Systems Chair Professor, Computer Science and Engineering (Joint appointment with Electrical Engineering), Indian Institute of Technology Delhi, New Delhi, India, srsarangi@cse.iitd.ac.in.

explicitly designed for this purpose. Depth images can be represented as a matrix of integers, where each cell represents the depth of the corresponding pixel in the color image. Various types of depth cameras exist, each based on a different working principle. Three of the most popular types are: LiDAR, structure light, and stereo camera.

Table 1. A comparison of various depth image acquisition methods

Aspect	Accuracy	Range	Res.	Adaptability	Affordability	Energy Efficiency	Compactness
LiDAR Sensors	High	High	Low	High	Low	Low	Low
Structured Light	High	Low	Medium	Low	Medium	Medium	Medium
Stereo Cameras	**High**	**Medium**	**High**	**Medium**	**High**	**High**	**High**



(a) Frame F_t (Color) (b) Noisy depth map N_t (c) Noisy depth map N_{t+1} (d) Corrected depth map D_t

Fig. 1. Visual representation of algorithmic holes (green ellipse) and flickering noise (yellow ellipse) across two frames. The flickering noise appears and disappears across frames, but algorithmic noise persists across frames as long as the object remains present.

As shown in Table 1, LiDAR and structured light sensors are much more expensive and as of 2024, it is quite challenging to implement them on lightweight autonomous systems like drones. Hence, price conscious mobile robot developers have increasingly favored stereoscopic depth cameras (or just stereo cameras) due to their compact size, reduced weight, low cost and the reasons mentioned in Table 1. A stereo camera such as Intel RealSense D455 or ZED 2 is a camera system that uses two or more cameras to capture images of the same scene from different perspectives to determine the depth by comparing the arrangement of pixels in both images — simulating the way human eyes perceive depth. This technique is known as *stereoscopic depth estimation*. Although stereo cameras provide depth images at lower cost with higher accuracy, the depth images generated are prone to imperfections such as noise and gaps. Furthermore, an erroneous perception of depth may result in unfavorable consequences such as collisions in tasks involving close contact [29], thereby causing damage to property and perhaps endangering humans [44, 46] (refer to Section 5.6.1). There are two type of inaccuracies present in the depth images generated using stereo cameras: ① flickering pixel noise and ② algorithmic holes that arise from failures in stereoscopic matching (refer to Fig. 1). The first type of noise is random in character [20] and may be corrected using classical spatial filters, whereas the second type requires far more complex solutions, this work provides one such solution.

Prior research in the domain of depth rectification can be broadly classified into two distinct categories: ① ML-based methods [27, 50, 54] and ② non-ML-based methods [6, 19, 22, 32]. There are two problems with ML-based methods, notably with methods that use neural networks. The first issue is that even the most optimized neural networks are computationally expensive and achieve only $2-3$ FPS on embedded hardware, $10\times$ slower than the required rate. The other major issue is the availability of a sufficient amount of real-world data for training. There are however, many proposed methods that expand the size of datasets by creating new artificial images based on the images that have already been collected [39, 42]. However, their applicability and generalizability are limited. Non-ML methods, on

the other hand, are faster compared to their ML counterparts and require only a minimal amount of precise ground truth data for calibration. The majority of non-ML techniques employ spatial [22] and temporal [6] filters on frames to eliminate noisy pixels. These approaches work well for removing flickering noise but perform poorly in the case of algorithmic noise. Given these trade-offs, this work aims to design a less computationally expensive (non-ML) depth rectification method tailored for embedded systems such as NVIDIA Jetson Nano [5, 25]. The goal is to achieve higher quality results compared to current state-of-the-art approaches. Therefore, the challenge is clear:[2]

Achieve high-quality depth rectification at a minimum of 20 FPS on a Jetson Nano, while keeping average power consumption under 5 watts.

In this paper, we introduce *VoxDepth*, a novel approach that is significantly different from prior depth rectification methods. To the best of our knowledge, *VoxDepth* is the first method to leverage a dynamic 3D point cloud representation to correct both flickering noise and algorithmic holes in depth images. This point cloud is repeatedly updated that accumulates geometric information across frames. We introduce several innovations in both the design of this point cloud representation and the strategies used to update it efficiently on embedded hardware. The fused point cloud is computed at the beginning of an epoch and it is expected to remain *stable* till the end of the epoch. Then we create a 2D template scene out of this "fused" point cloud. Whenever a new 2D image arrives, we compare it with the 2D template and identify the regions that may have suffered from any kind of noise. Algorithmic holes are corrected using information derived from the template, whereas flickering noise is corrected using standard filters. When the environment sufficiently changes, a new epoch begins, and the fused point cloud is computed yet again. This novel method allows us to sustain a frame rate of 27 FPS (at least 70% more than prior work) and outperform the state-of-the-art in terms of depth estimation quality. The novel aspects of the design of *VoxDepth* are as follows:

① A point cloud fusion method that aims to combine depth information from a set of consecutive RGB-D frames into a single sparse point cloud.

② A depth image inpainting method that is used to create a high-resolution depth image from a low-resolution point cloud; it is meant to be used as a 2D scene template.

③ A pipelined module for combining the foreground and background to produce precise 2D depth images from the captured depth image. This process involves (a) resizing the images, (b) estimating motion, (c) correcting the incoming frame with the help of the 2D scene template in order to produce depth images that exhibit a high level of accuracy.

④ A technique to dynamically recompute the point cloud when the scene changes to a sufficient extent.

The paper is organized as follows. We begin with a detailed background and related work in the domain of depth rectification (Section 2). Section 3 discusses the motivation for the work. Section 4 describes our two-step depth image rectification approach. To support our claims, we evaluate our design in Section 5. We finally conclude in Section 6. The code for the project is available at the following git repository https://github.com/Akanksha-Dixit/Voxdepthcode.git.

2 BACKGROUND AND RELATED WORK

Over the past few years, there have been several notable attempts to solve the problem of depth rectification by exploiting spatial and temporal similarities in depth frames. Ibrahim et al. [20] present a comprehensive list of many such methods in their survey paper. We also present a brief comparison of the most relevant related work in Table 2. Previous work in

[2]These performance and power constraints align with prior studies [4, 16].

this field can be classified based on the following features: utilization of point clouds, motion compensation, the usage of RGB channels in depth correction and the type of the technique (ML or non-ML).

Table 2. A comparison of related work

Year	Work	Method	ML-based	Color-Guided	Point Cloud	Motion Comp.	Setup @ Frame Rate (fps)
2011	Matyunin et al. [32]	Motion Flow Temporal Filter	×	✓	×	✓	Intel Celeron 1.8 GHz CPU @ 1.4
2016	Avetisyan et al. [6]	Optical Flow Temporal Filter	×	✓	×	✓	Workstation GPU @ 10
2018	Grunnet-Jepsen et al. [19]	Spatial Filter	×	×	×	×	Jetson Nano @ 39.52
2018	Islam et al. [22]	Gradient & LMS-based Filter	×	×	×	×	Jetson Nano @ 16.89
2019	Sterzentsenko et al. [42]	Multi-sensor Guided Training	✓	×	✓	×	GeForce GTX 1080 GPU @ 90
2019	Chen et al. [14]	2D-3D Feature-based	✓	✓	✓	×	Not Available
2021	Senushkin et al. [39]	Decoder-Modulation CNN	✓	✓	×	×	NVIDIA Tesla P40 GPU
2021	Imran et al. [21]	Surface Reconstruction Loss	✓	✓	×	×	GeForce GTX 1080 GPU Ti @ 90
2023	Krishna et al. [27]	Temporal Encoder	✓	✓	×	✓	Jetson Nano @ 2
2024	**VoxDepth**	**3D Filter**	×	✓	✓	✓	**Jetson Nano @ 26.71**

2.1 ML-based Methods

ML methods have proven to be quite accurate in such tasks, however, they are not tailored for real-time embedded devices as of today. Moreover, there is a paucity of real-world datasets and as mentioned the algorithms in use are often quite slow and power consuming. They are thus not suitable for edge devices like the Jetson Nano board. Sterzentsenko et al. [42] show that the issue of lack of ground truth data can be tackled by having multiple sensors capture the same scene (multi-sensor fusion). This allows the system to create its own semi-synthetic dataset by first creating a highly accurate 3D representation of the scene by fusing the information from multiple viewpoints, and then generating a series of 2D or 3D scenes. This method is effective in small enclosed spaces, but it cannot be implemented in open outdoor spaces because of the large number of sensors required (each of which would cost over $400 USD) and the other environmental limitations traditionally associated with an outdoor environment. Most ML approaches [27, 50, 54] use an encoder-decoder CNN architecture similar to Unet [36]. The encoder is responsible for feature extraction whereas the decoder uses those features to improve the original depth map. Some works like DeepSmooth [27], also suggest the use of two encoder branches: one for color and the other for depth images to train the network. While this technique produces highly accurate depth images, we have shown in Section 5.3.1 that this method is slow (least performing in our evaluation). The frame rate is way lower than 20 FPS (our minimum threshold). Our method differs form existing works in both objective and implementation. We use a lightweight, voxel-based 3D representation without relying on deep learning, making it suitable for embedded systems with latency and memory constraints. Unlike previous works [14, 42], which ignore motion information, *VoxDepth* incorporates RGB-D odometry to handle fast motion and avoid ghosting.

2.2 Non-ML-based Methods

Non-ML methods tend to generalize better than their ML counterparts, which are heavily dependent on the amount, diversity and quality of the baseline datasets. The bulk of the work in this domain relies on classical filters. The core idea is to detect and flag erroneous pixels by identifying atypical features (both temporal and spatial). These flagged pixels are then replaced by predicted depth values. Grunnet et al. [19] only use the spatial neighborhood of the depth image to fix holes in the frame (a spatial filter). This proves to be the least computationally expensive approach, and hence ships with the Intel RealSense SDK. But these methods fail to perform as we have observed in Section 5.2. Few approaches in this domain [6, 32], use previous frames in the depth videos to both detect and replace erroneous pixels. This is a temporal filter. To account for motion within the frame as well camera movement, they also suggest compensating the motion with methods such as optical flow. This results in a considerable amount of latency on smaller devices. Islam et al. [22] combine the spatial and temporal filters in their work to filter out erroneous pixels. They do not, however, suggest any motion compensation in favor of faster processing. This results in a considerable hit in terms of quality when compared to *VoxDepth* (refer to Section 5.2).

2.3 Image Matching

In Section 2.2, we discussed that state-of-the-art depth rectification methods often rely on image matching algorithms to detect erroneous pixels. In this section, we provide a detailed explanation of the image matching process. Image matching involves finding *correspondences* between two or more images of the same scene or object. The goal is to identify points or regions in one image that match corresponding points or regions in another image. This process typically involves three fundamental stages:

(1) **Feature Detection** involves identifying distinctive key points or features in the images. Two widely used methods for feature detection are SURF [7] and FAST [48].
(2) **Feature Description** creates descriptors for the detected features, which are robust to changes in scale, rotation, and illumination. The most commonly used feature descriptor is the BRIEF [11] descriptor.
(3) **Feature Matching** uses the features with their accompanying descriptors to find correspondences. This is a time-consuming task and brute force solutions might lead to a sluggish system. Hence, approximate matchers such as FLANN [35] are used to speed up the process.

2.4 Stereoscopic Depth Estimation

The concept of stereoscopic depth estimation [28] involves employing two distinct sensors (cameras) to observe the same scene from two different angles. The distance between the two sensors is known and is referred to as the *baseline* (b). They are positioned on the same plane and synchronized to capture a scene simultaneously. Each pixel in the frame captured by the first sensor, C_l, is then matched with its counterpart in the frame captured by the other sensor, C_r. For a point in the scene, say P with coordinates (x, y, z) in the camera's frame of reference, each sensor would see a projection in its respective image plane. The distance (in pixels) between a point P_l in the image plane of sensor C_l and its corresponding point in the image plane of C_r (P_r) is referred to as the *disparity* (refer to Fig. 2). This is the transform that maps each pixel in one image to the corresponding pixel in the other image. We only consider the x coordinates because the y coordinates are the same for parallel sensors. Using the property of similar triangles between triangles ΔPP_lP_r and ΔPC_lC_r, we arrive at the following relation:

$$\frac{b}{z} = \frac{b + x_l - x_r}{z - f} \text{ Hence, } z = \frac{f \times b}{(x_r - x_l)} \tag{1}$$

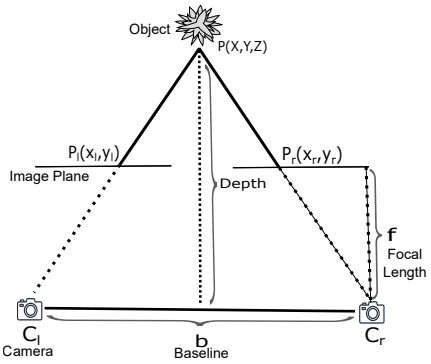


Fig. 2. Visual representation of the stereoscopic depth estimation method

Here z is the depth of the point P, f is the cameras' focal length and b is the baseline. Stereo cameras such as Real Sense D455 use this technique to compute depth images. These depth images contain substantial noise in the form of *algorithmic* and *flickering* noise (refer to Section 2.7).

2.5 RGB-D Odometry

The objective of RGB-D odometry is to estimate the rigid body motion q of the camera given two consecutive images (the source and the destination). This *motion* is a transformation with six degrees of freedom: three describe camera rotation (roll, pitch and yaw) and three represent translation (x, y, z). Using this transformation, we can align multiple point clouds to a common coordinate system. Then, we can fuse them together to form a *dense point cloud*, which is required in *VoxDepth*.

2.6 Image Registration

Image *registration* refers to the process of aligning two distinct images or frames of a shared scene to a common coordinate system. In order to accomplish this, we calculate the affine transformation between two frames by comparing the pixels in the two frames. An affine transformation refers to a geometric transformation that preserves points, straight lines and planes. It includes operations such as translation (shifting), rotation, scaling (resizing) and shearing (stretching). The affine transformation can be represented as a rotation A (multiplication by a matrix) followed by a translation b (scalar addition). Hence, the transformed coordinates of the image are $P_1 = A \times P_0 + b$. Here, P_0 represents the original coordinates of the pixel, and P_1 represents the computed (transformed) coordinates. In our implementation, we condense this transformation using a single 3×3 matrix M. To estimate an affine transformation, the squared difference between the real image and the transformed image is minimized. Formally, if an image is represented as $I(u, v)$, denoting the pixel intensity at the pixel coordinates (u, v), then we minimize the following objective function:

$$E = \sum_{u,v} \left\| I_1 \left(M \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} \right) - I_0 \left(\begin{bmatrix} u \\ v \end{bmatrix} \right) \right\|^2 \tag{2}$$

Here, I_0 and I_1 are consecutive frames. Image registration aids in estimating the motion of the camera and transforming frames compensates for that motion. It brings frames captured at different points of time to a common coordinate system.

2.7 Algorithmic Noise and Flickering Noise

Algorithmic noise refers to the presence of inaccurate regions in depth images resulting from the inability to locate corresponding pixels captured by the other camera. Ibrahim et al. [20] use terms like "hole pixels" and "spatial and temporal artifacts" to describe algorithmic noise. We have categorized all these artifacts under a single term based on the cause of this noise. This can happen due to a couple of reasons: ① occlusion of the point in one of the cameras; or ② lack of textural information leading to patches of invalid pixels, which appear as red shadows in the depth images in Fig. 1 (refer to the green ellipse). Flickering noise in depth images refers to rapid and irregular variations in pixel intensity, often appearing as random fluctuations or shimmering effects in the image (refer to the yellow ellipse in Fig. 1). This type of noise is commonly caused by sensor imperfections such as fluctuations in the sensitivity or exposure time leading to inconsistent depth readings and pixel values. These noise artifacts can be corrected by local filters. Some of it can also happen due to camera motion.

2.8 3D Point Cloud & Fusion

2.8.1 Point Cloud. Point clouds are 3D data structures composed of points representing the surface of objects or scenes in the real world. Let us start with mathematically representing the RGB-D frames generated by a stereoscopic depth camera. I_{RGB} represents the channel-wise RGB intensity and I_D is the depth of the point in meters in the image plane. From the depth image I_D, we can compute the surface S ($S : \gamma \rightarrow R^3$) visible from the sensor by projecting the pixels in 3D space for each point $p = (p_x, p_y) \in \gamma$, $\gamma \subseteq \mathbb{R}^2_+$ in the depth image at time t (see Equation 3).

$$\begin{aligned} I_{RGB} &: \gamma \times \mathbb{R}_+ \rightarrow [0,1]^3, (p,t) \rightarrow I_{RGB}(p,t) \\ I_D &: \gamma \times \mathbb{R}_+ \rightarrow \mathbb{R}_+, (p,t) \rightarrow I_D(p,t) \\ S(p) &= \left(\frac{(p_x + o_u) I_D(p,t_0)}{f_x}, \frac{(p_y + o_v) I_D(p,t_0)}{f_y}, I_D(p,t_0) \right)^T \end{aligned} \tag{3}$$

Here, $(o_u, o_v)^T$ is the principal point of the camera on the $xy-$coordinate system. The *principal point* refers to the point on the image plane where the line passing through the center of the camera lens (optical axis) intersects the image plane. The focal lengths in the x and y directions are represented by f_x and f_y, respectively. Each point in the *point cloud* corresponds to a specific position in space and is specified by its x, y and z coordinates. To combine multiple such point cloud data structures, we need to transform them to a *common coordinate system* first. This is done using RGB-D odometry as described in Section 2.5.

2.8.2 Point Cloud Fusion. Point cloud *fusion* is the process of aligning and merging point clouds from multiple viewpoints (or sensors) to build a more complete and accurate 3D model of the scene. RGB-D odometry is used to align multiple point clouds to a common coordinate system. In our method, we add an additional step that involves *reprojecting* the merged point cloud back to a 2D image to create the (template), by following the reverse sequence of steps (reverse of Equation 3). This fusion process creates a point cloud that is resilient to effects of occlusion (in one of the cameras) and loss of some textural information. This is because it is computed over a window of time and thus any loss of data in individual depth images is compensated for. Note that for this process to work, the object

mounting the stereo camera has to be in motion, which is most often the case. Because of the motion the same depth image is captured from slightly different angles. After odometry and fusion, we arrive at a more robust 3D description of the scene captured in our fused point cloud. Note that the window of time cannot be very long though. Then the scene will change drastically and so will the point cloud. Find the optimum length of the window based on empirical considerations is a part of our contributions.

3 CHARACTERIZATION AND MOTIVATION

3.1 Overview

In this section, we shall discuss our experimental setup and datasets used, as well as some experiments to motivate the problem statement and support the methodology. We want to characterize two specific aspects of our setup: ① The impact of *algorithmic noise* on the depth image quality, and ② The efficacy of *motion compensation* methods as compared to a state-of-the-art method namely *optical flow*. For designing our system, these were our most important decision variables.

3.2 Experimental Setup

We designed our system for the NVIDIA Jetson Nano B01 development Kit, a compact and affordable single-board computer created by NVIDIA. This board is specifically tailored for AI applications, machine learning, robotics, and edge computing. The technical specifications of our setup are shown in Table 3.

Table 3. Jetson Nano specifications

Processing	
GPU	NVIDIA Maxwell architecture with 128 NVIDIA CUDA® cores
CPU	Quad-core ARM Cortex-A57 MPCore processor
Memory	
Memory	4 GB 64-bit LPDDR4, 1600MHz, 25.6 GB/s
Storage	16 GB eMMC 5.1
Camera and Connectivity	
Camera	12 lanes (3x4 or 4x2) MIPI CSI-2 D-PHY 1.1 (1.5 Gb/s per pair)
Connectivity	Gigabit Ethernet
Mechanical	
Mechanical	69.6 mm x 45 mm, 260-pin edge connector

Table 4. Intel RealSense camera specifications

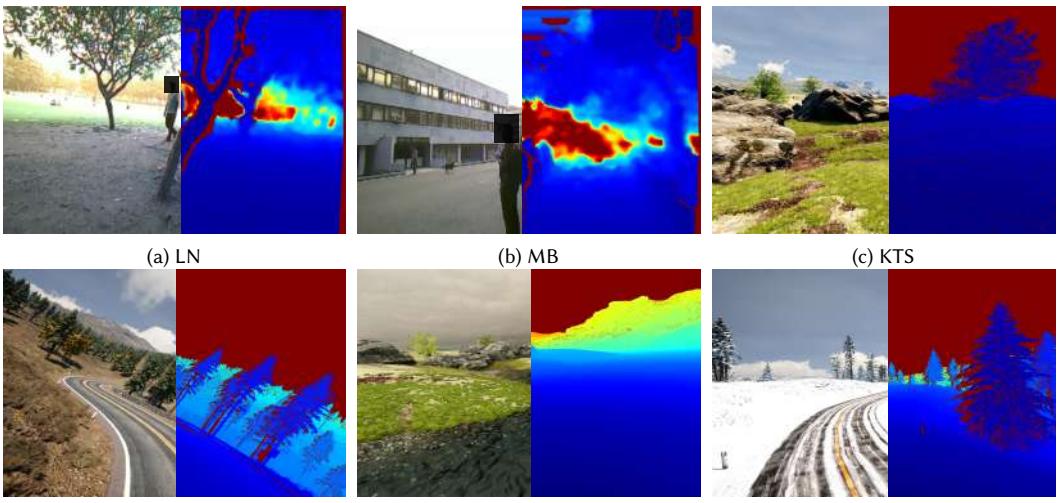
Operational Specifications	
Depth Accuracy	< 2% at 4m
Depth Resolution and FPS	1280x720 up to 90 FPS
Depth Field of View	86° × 57°
Components	
RGB Sensor	Yes
Tracking Module	Yes
Module Specifications	
Dimensions	124mm x 29mm x 26mm
System Interface Type	USB 3.1

3.3 Overview of the Dataset

In this paper, we have taken two sets of datasets: real-world and synthetic. We obtain our real-world depth images (RGB-D) using the Intel RealSense D455 depth camera [24]. To address the shortage of real-world outdoors depth datasets employing stereo cameras, we created our own dataset with manually derived ground truths. For the technical specifications of the camera refer to Table 4. We created the dataset by shooting at two different locations.[3] We capture the following data: dense 16-bit integer depth images, a stereo pair of colored images, IMU (Inertial Measurement Unit) and accelerometer data. Finally, we manually created a set of ground truth images from the raw depth images to evaluate our methods. Moreover, we also used synthetically generated images from the Mid-air [18] dataset, which contains data from several trajectories of a low-altitude drone in an outdoor environment in four different weather conditions generated using the Unreal game engine. The specific datasets used in this work are shown in Table 5 and example frames are shown in Fig. 3.

Table 5. Datasets used in this work

Abbr.	**Name**	**Depth Dimensions**	**Acquisition Method**
LN	Lawns	360x640	RealSense
MB	Building	360x640	RealSense
KTS	Kite Sunny	1024x1024	Unreal Engine
KTC	Kite Cloudy	1024x1024	Unreal Engine
PLF	Procedural Landscape Fall	1024x1024	Unreal Engine
PLW	Procedural Landscape Winter	1024x1024	Unreal Engine



(a) LN (b) MB (c) KTS

(d) PLF (e) KTC (f) PLW

Fig. 3. Depth images from different datasets used in this work

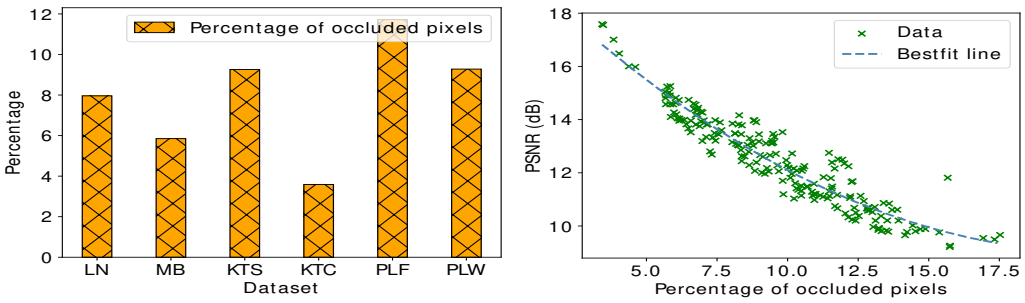
[3]The anonymity of the individuals depicted in the dataset photographs, as well as the scenes, has been safeguarded by making the faces indiscernible.

3.4 Impact of Algorithmic Noise

In Section 1, we discussed that state-of-the-art methods are generally more effective at mitigating flickering noise than algorithmic noise, so we need a new method to improve that. However, one may ask whether the algorithmic noise is important. This subsection aims to evaluate its impact more concretely. As discussed in Section 2, occlusions lead to algorithmic noise. Fig. 4a shows that our dataset has a varying degree of occluded patches. In Fig. 4b, we see that the quality of the generated depth images, measured in PSNR (or Peak-Signal to Noise Ratio) decreases significantly as the percentage of occluded pixels in the depth image increases. We observe a superlinear decrease in the image quality as we increase the percentage of occluded pixels. For a 10% increase in the number of occluded pixels, we see a 28.1% reduction in quality in terms of PSNR.

3.5 Relevance of the State-of-the-art: Optical Flow based Techniques

An extremely popular method of depth correction is motion compensation using optical flows [6, 32]. An *optical flow* enables the system to predict the movement of specific objects in the scene at the pixel level, making it amenable to motion correction and to certain extent image registration. Here, we outline our arguments opposing the utilization of optical flows. Although several contemporary embedded systems such as Jetson AGX Orin [1] possess a built-in optical flow accelerator, but they are limited by other shortcomings such as high power consumption, weight and cost. The Orin has a price tag of approximately $\$2,000$ USD and a weight of approximately 1.4 kg. This exceeds the constraints that we outlined in Section 1. We tested the RAFT optical flow network [43] with the necessary optimizations and a CUDA implementation of the Horn-Schunck optical flow technique [33] on our Jetson Nano board. We observed that RAFT network requires 15.5 seconds to perform one inference, while the Horn-Schunck approach takes 2.45 seconds to compute the optical flow for a single frame. Both of these latencies significantly exceed the acceptable threshold for a real-time system, where each frame needs to be computed/processed within 50 milliseconds (20 FPS). This proves that optical flow based systems are not a suitable choice for real-time applications of this nature.



(a) Average percentage of incorrect pixels in depth images across datasets

(b) Percentage of occluded pixels vs PSNR (dB)

Fig. 4. Plots depicting (a) the ratio of inaccurate pixels in different datasets and (b) the effect of occlusion holes on the quality of depth images – PSNR of the raw depth image (vis-a-vis the ground truth)

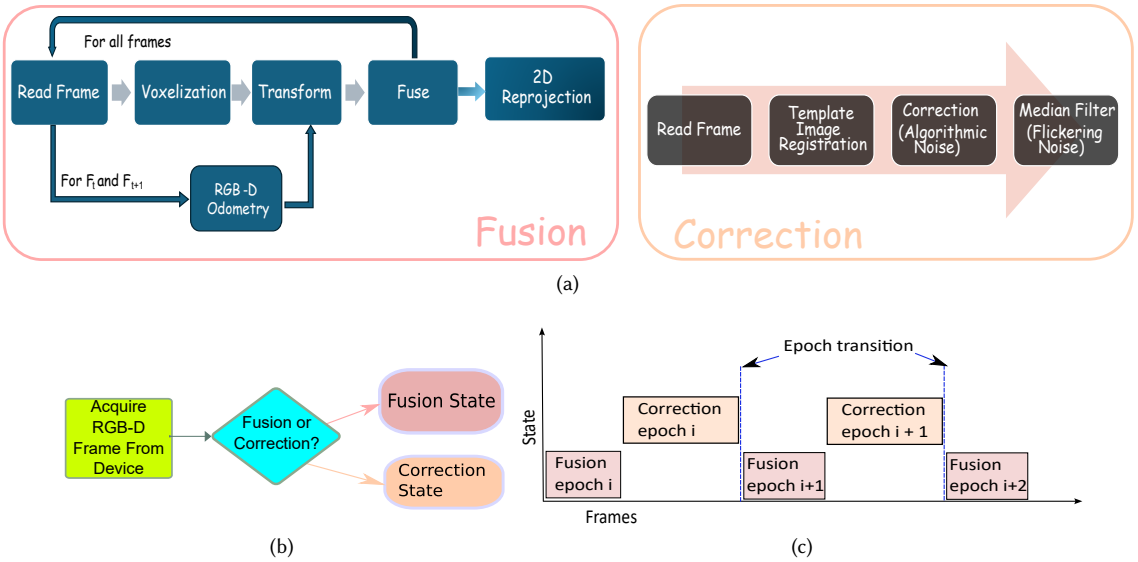


Fig. 5. (a) Flow diagram of the proposed method. The two states of the system have been encased in labeled boxes. The left box, the *fusion* state, is responsible for creating an accurate representation of the scene in the form of a template image. The right box, named the *correction* state uses the template image generated in the last state to fix the inaccuracies in the raw depth images. (b) The conditional switching between the states is explained in Section 4.4. (c) A timing diagram showing how the epochs proceed.

Insights

① There is a super-linear decrease in the depth image quality with an increase in the percentage of occluded pixels.
② Optical flow based motion compensation techniques are too slow to be implemented on real time embedded systems applications.

4 IMPLEMENTATION

4.1 Overview

Our objective is to rectify the erroneous patches in the raw depth images and eliminate sensor noise, while also achieving a target frame rate of 20+ FPS. The goal of the system is to exploit spatio-temporal similarities in a scene in three dimensions to fill the gaps in the captured depth images. To achieve a high frame rate, the system needs to run high-latency components as infrequently as possible while also being adaptive and responsive to scene changes. In this paper, we propose an epoch-based method for correcting noisy depth images. Each epoch has two main phases: fusion and correction, as shown in Fig. 5a. In the fusion phase (Section 4.2), a series of depth frames is combined to produce a single fused image. This fused image then serves as a reference in the correction phase (Section 4.3), where it is used along with the noisy depth image to generate a corrected output frame. The system automatically switches between the fusion and correction phases using an epoch transition module, illustrated in Figures 5b and 5c. This module monitors the quality of image registration and determines when to switch phases. We describe this module in detail in Section 4.4.

4.2 Point Cloud Fusion

The first step is to use the first n frames ($F_1 \dots F_n$) to create a *fused* point cloud that represents an accurate 3D representation of the scene. Here, n is the fusion window size (studied in Section 5.5.1). Note that our method needs to ensure the following properties:

① It must not exceed the limited 4 GB memory available on the embedded device.

② It needs to run in parallel and use all the concurrent resources of the in-built GPU.

4.2.1 Voxelization. To fuse incoming depth frames (2D images), we first transform each incoming color-depth image pair (RGB-D image) into a point cloud (3D representation) using Equation 3. Before we can design a fast strategy for fusing point clouds, we need to manage the application's memory footprint. Storing all the points in a fused point cloud can lead to high latencies and inefficient use of memory. Also, the distance between successive points in the point cloud is variable – this makes it hard to process it. We thus store the point cloud using a discrete three-dimensional grid data structure called a *voxel grid*, where the constituent points (voxels) represent an equal-sized cubic volume. They are uniformly spaced. This conversion process is referred to as *voxelization*. Note that in a voxel grid, each point is associated with a Boolean value; if it is 1, then it means that the corresponding point exists in the point cloud, and vice versa. A voxel grid is thus a sparse 3D matrix. Since each pixel in the depth image can be processed independently, we implement a CUDA [30] based parallel algorithm as described in Algorithm 1 to implement voxelization. Fig. 6a shows an example of a voxelized point cloud.

Algorithm 1 Voxelization algorithm

Input:

`CX,CY,FX,FY` ▷ Camera intrinsic params
`GridSize,XVoxSize,YVoxSize,YVoxSize` ▷ Voxel params
`imgW,imgH` ▷ Image dimensions

1: **function** toSurfaceCaller ▷ CUDA kernel function
2: `xIndex` ← ThreadIdx.x ▷ One thread per input pixel
3: `yIndex` ← ThreadIdx.y
4: **if** (`imgH,imgW`) within image limits **then**
5: `z` ← `flImgD[xIndex][yIndex]`
6: **if** `z` ≠ 0.0 **then** ▷ Calculate voxel coordinate for pixel
7: $xVox \leftarrow \text{floor}\left(\frac{((\texttt{xIndex}-\texttt{CX})\times \texttt{z})}{\texttt{FX}\times\texttt{XVoxSize}}\right) + \frac{\texttt{GridSize}}{2}$
8: $yVox \leftarrow \text{floor}\left(\frac{((\texttt{yIndex}-\texttt{CY})\times \texttt{z})}{\texttt{FY}\times\texttt{YVoxSize}}\right) + \frac{\texttt{GridSize}}{2}$
9: $zVox \leftarrow \text{floor}\left(\frac{\texttt{z}}{\texttt{ZVoxSize}}\right)$
10: **if** (`xVox,yVox,zVox`) inside voxel **then**
11: `voxelGrid[xVox][yVox][zVox]` ← 1
12: **end if**
13: **end if**
14: **end if**
15: **end function**

4.2.2 RGB-D Odometry. Next, we fuse the voxelized point clouds. However, to fuse the voxelized point clouds from multiple frames, we first need to align them, since they are captured from different viewpoints. For this, we use a visual odometry technique described in Section 2.5, based on the method proposed by Steinbrucker et al. [41], to estimate the

(a) Voxelized Point cloud from a single depth image (b) Fused point cloud

Fig. 6. Fused point cloud generated after the fusion step. The fusion process fills up holes in the point cloud.

camera motion. This motion is calculated once at the start of each *epoch*, using the first two frames, F_t and F_t+1. The motion computed between these two frames is referred to as the *transform*. To reduce computational overhead and maintain low latency, we assume that the camera's velocity remains nearly constant during the fusion window. Therefore, we approximate the motion between all consecutive frames in the window using this single transform.

4.2.3 Fusion. The final step comprises the fusion process. The fusion process uses a Boolean OR operation to determine if a voxel is occupied. A Boolean OR between two voxel grids is defined as a voxel-wise OR operation. If $V_{(i,j,k)}$ represents a voxel at a discrete location (i, j, k), then the Boolean OR ($\vee$) between two voxel grids V_1 and V_2 with dimensions $N \times N \times N$ can be defined as:

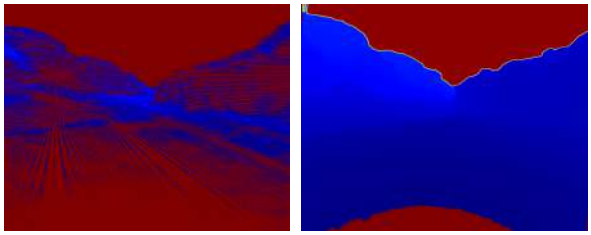
$$\begin{aligned} V_1 \vee V_2 &= \{V_{1(i,j,k)} \; OR \; V_{2(i,j,k)}; 0 \leq i, j, k < N, \forall i, j, k \in \mathbb{Z}\} \\ PC_{final} &= f^T(...f^T(f^T(PC_t) \vee PC_{t+1}) \vee ....) \vee PC_{t+n-1} \end{aligned} \tag{4}$$

The final fused voxel point cloud PC_{final} is generated by successively transforming and integrating new point clouds. Here f^T is the transform function (estimated in Section 4.2.2). $\vee$ is the Boolean OR operation between voxel grids. Fig. 6b shows an example of a fused point cloud.

4.2.4 2D Projection. The fusion process of raw 2D depth images results in a *fused* 3D voxelized point cloud. Subsequently, this fused voxelized point cloud shown in Fig. 6b is projected back to a 2D 16-bit integer depth image. This 2D *template depth image* is used to correct the raw 2D depth frames. The 2D depth image generated after fusion is most often too sparse for it to be used in the correction step (refer to Fig. 7a). A sparse template image would lead to misidentification of depth values in the raw frame. There are local [26] and non-local [10] filter-based methods for inpainting 2D depth images. We chose a class of local techniques called *morphological transformations* [47] to perform the inpainting due to their simplicity and parallelizability. Specifically, *grayscale dilation* (see Equation 5) is a morphological operation where each pixel in a grayscale image is replaced by the maximum in its neighborhood. Similarly, *erosion* (Equation 6) is a morphological operation where each pixel in a grayscale image is replaced by the minimum in its neighborhood. The final inpainted image is shown in Fig. 7b.

$$\begin{aligned} I_{transform}(x, y) &= \max\left(I\left(i, j\right)\right) \\ \text{where, } i &\in [x - \lfloor N/2 \rfloor, x + \lfloor N/2 \rfloor]; \\ j &\in [y - \lfloor N/2 \rfloor, y + \lfloor N/2 \rfloor] \end{aligned} \tag{5}$$

$$\begin{aligned} I_{transform}(x, y) &= \min\left(I\left(i, j\right)\right) \\ \text{where, } i &\in [x - \lfloor N/2 \rfloor, x + \lfloor N/2 \rfloor]; \\ j &\in [y - \lfloor N/2 \rfloor, y + \lfloor N/2 \rfloor] \end{aligned} \tag{6}$$



(a) Sparse fused depth image (b) Inpainted depth image

Fig. 7. Inpainting 2D projections of a fused point cloud to create a dense-scene representation

This template depth image is used in the depth correction step to fill in the incorrect patches in the incoming raw depth images from the stereo camera. Once we have our *template*, the first part of the *epoch* is complete, and we move on to the *correction* state.

Fig. 8 shows the complete fusion process by which a fused point cloud is created using the first n frames at the beginning of an epoch.

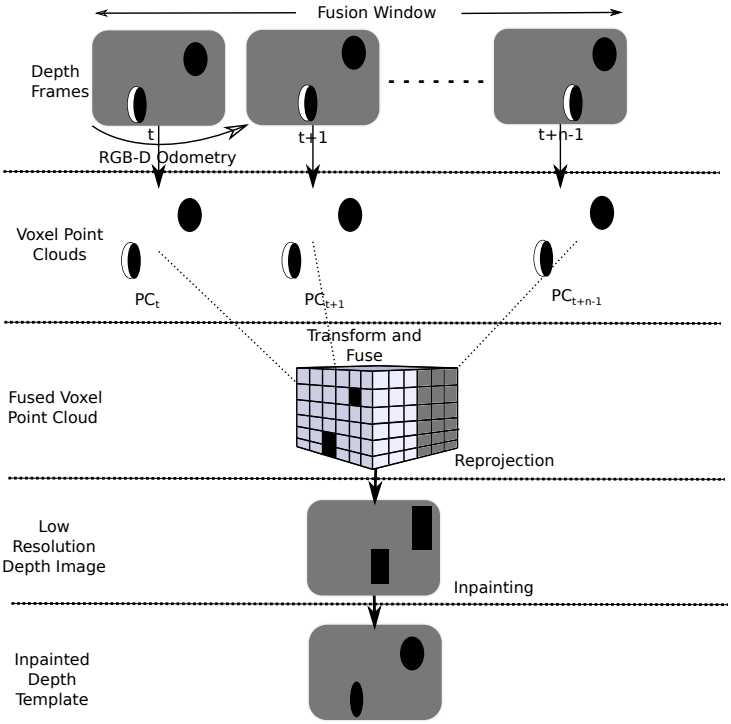


Fig. 8. Visual representation of the fusion process (Section 4.2)

4.3 Depth Correction

The final stage of the process, depth correction, is detailed in this section. The following are a few of the challenges we encountered while developing a fast depth correction step:
❶ The correction must take into account the robot's motion in order to correctly rectify the values of the inaccurate pixels.
❷ The latency may increase significantly due a bottleneck caused by reading frames from the device (I/O operations). Considering these factors, we created a motion-aware correction module that is pipelined and employs all of the system's concurrent hardware.

4.3.1 Template Image Registration. Let the *template* be I_T. When we read a new frame I_F, we combine it with the template to create the corrected frame. Before we combine the new frame and the template, we must first transform I_T to the coordinate plane of the current frame. This is achieved by estimating the affine transform described in Section 2.6. We use the *ORB* (Oriented FAST and Rotated BRIEF) feature matching algorithm introduced by Rublee et al. [37] to match features across a pair of frames and estimate the transform. Since estimating the transform is a time-consuming task, as we saw in Section 5.5.5, we downscale the RGB frames to 200×200 pixel frames, and then estimate the transform.

4.3.2 Algorithmic Noise Correction. Finally, the transformed template is combined with the foreground depth by replacing the inaccurate pixels in the new image with the corresponding pixels in the template.

$$I_C = \{I_F[i,j] \mid \text{if } I_F[i,j] \text{ is valid; else } M(I_T[i,j])\} \tag{7}$$

Here, I_C is the combined final image and $M(I_T)$ is the transformed template image. Note that a pixel is considered *inaccurate* if either ❶ its value is less than half of the corresponding pixel's value in the template $M(I_T)$ or ❷ the pixel's value is greater than the corresponding pixel's value in $M(I_T)$.

4.3.3 Flickering Noise Filter. We apply a median filter across the combined depth image (I_C) to filter out flickering noise by exploiting the spatial similarities in it. A median filter is a spatial filter that replaces a pixel value by the median of its neighborhood. It is effective in removing 'salt and pepper'type of noise from images while still preserving edges. It is most often implemented by moving a square window W centered at a pixel (x, y) across the image. The median of all the values in the window is used to replace the pixel at (x, y). Mathematically, for an image defined by the function $I(x, y)$ (=pixel intensity at (x, y)), a median filter can be defined as follows (square $N \times N$ window):

$$\begin{aligned} I_{filtered}(x,y) = \text{median}\,(I\,(i,j)) \text{ where} \\ i \in [x - \lfloor N/2 \rfloor, x + \lfloor N/2 \rfloor]\,; \\ j \in [y - \lfloor N/2 \rfloor, y + \lfloor N/2 \rfloor] \end{aligned} \tag{8}$$

4.3.4 Correction Pipeline. To effectively use all the resources available to us, we convert the aforementioned depth correction technique into a software pipeline (refer to Fig. 9) that uses multi-threading. We identify three different operations that have contrasting requirements. These are as follows:
① Reading frames from the sensor, mostly an I/O bound operation.
② Estimating the transform, implemented on the CPU.
③ Combining step, implemented on the GPU.

We propose a three-stage pipeline to implement these as separate stages to improve the overall efficiency of the system. We present a visual summary of VoxDepth including pictorial representations of the outputs of each phase in Fig. 10.

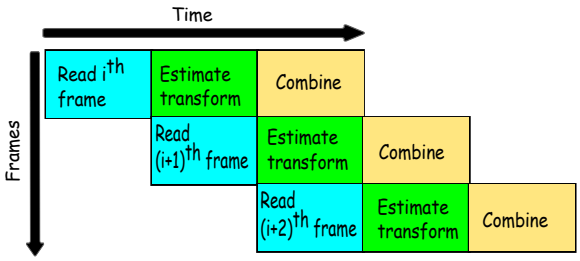


Fig. 9. Software pipeline used to ensure the efficiency of the depth correction method (Section 4.3)

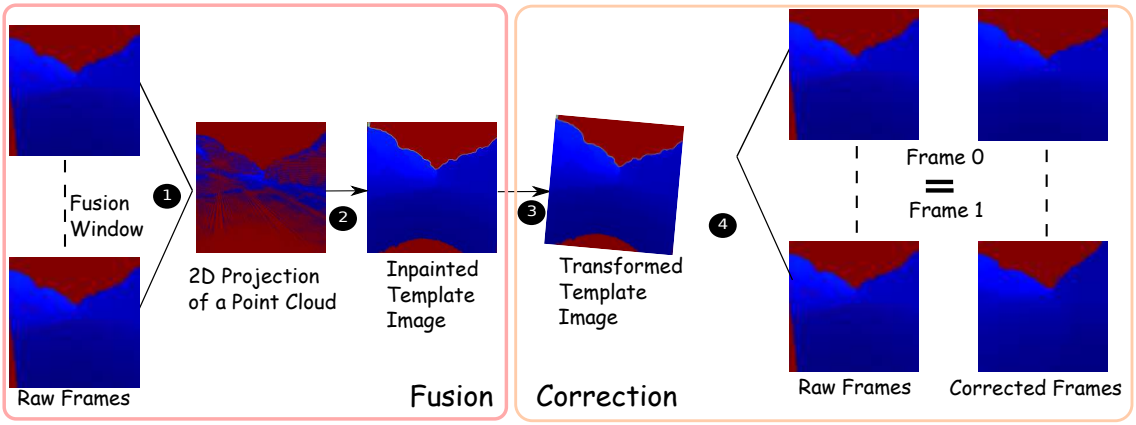


Fig. 10. A visual representation of the proposed method with images representing outputs from each of the major steps. (1) First, the frames in the fusion window are fused together to form a fused *voxel grid*, which is projected back to 2D, (2) The sparse projection is inpainted to create a dense *template image*, (3) The template is transformed to align with the current frame and (4) Finally, the incoming raw frame is combined with the template to generate the corrected depth image.

4.4 Epoch Transition

After implementing both of our methods, it is necessary to dynamically recalculate the point cloud after the scene has undergone significant changes. At this point, it is thus necessary to initiate a new *epoch*. We want to perform point cloud fusion as infrequently as possible since it is the slowest component. The conditional box shown in Fig. 5b refers to this decision process. When the scene undergoes significant changes, our previous template becomes unsuitable for the current scene, necessitating the initiation of a new *epoch*. We observed that when there is a significant shift in the scene since the previous fusion step, the quality of the matching results deteriorates significantly. This phenomenon is studied in Section 5.5.3. In our approach, we employ a feature matching technique based on the ORB method [37]. ORB uses the widely recognized BRIEF [11] feature descriptors in its matching method. These feature descriptors are vectors that provide a description of the features in an image, which are then utilized for the purpose of matching. In order to assess the quality of a match, we calculate the number of matches where the *feature* distance (Euclidean distance)

between them exceeds a threshold (= 20), a value that we determined experimentally. We call such matches as *good matches*. The number of good matches between a pair of images is computed and if this value is below a predetermined threshold (5 in our experiments), the system transitions back to the fusion state, initiating a new epoch. This process is described in the timing diagram shown in Fig. 5c.

5 RESULTS AND ANALYSIS

① First, we present a visual summary of *VoxDepth* including pictorial representations of the outputs of each phase in Fig. 10.
② To evaluate the visual quality of the proposed method, VoxDepth, we conduct a comprehensive comparison with various state-of-the-art depth rectification techniques.
③ To assess the feasibility of depth correction algorithms on embedded devices, we analyze their runtime performances.
④ In Section 5.5, we show two experiments conducted on *VoxDepth* to understand how the size of the fusion window affects image quality and how we developed the necessary conditions for a state switch.
⑤ A simulated study is also presented to demonstrate the impact of noise in depth estimation in a drone swarming task.
For all these experiments, we use the same experimental setup and datasets thoroughly described in Section 3.

5.1 Quality Metrics

5.1.1 PSNR. The quality of depth maps is measured by calculating the peak signal-to-noise ratio (PSNR) between the generated depth images and their corresponding ground truth depth images. The PSNR is computed using the mean squared error (MSE) between corresponding pixels in the ground truth and corrected images. It is further normalized to the maximum possible pixel value (typically 65,535 for 16-bit depth images). This normalized value is scaled to a logarithmic scale to represent the values in decibels. Higher PSNR values indicate better image quality. A perfect reconstruction corresponds to an infinite PSNR.

5.1.2 Masked RMSE Metric. By utilizing the masked RMSE metric proposed by Cao et al. [12], we try to evaluate and compare the depth correction techniques in terms of their capacity to fill the areas that are occluded in one of the cameras. Specifically, we compute the root mean squared error (RMSE) between the pixels of the ground truth depth images and the corrected ones only for the occluded regions.

5.2 Comparison with State-of-the-art Depth Rectification Methods

In this section, we compare our proposed method, *VoxDepth*, with three state-of-the-art methods: an ML-based method (DeepSmooth [27]), a non-ML algorithm (GSmooth [22]), and a commercial solution (Intel RealSense's built-in hole-filling [19]). DeepSmooth was trained on the Kite Sunny (KTS) dataset using the hybrid loss function from the original paper [27]. GSmooth, the non-ML baseline, uses the least median of squares in both spatial and temporal domains to identify outliers. For the commercial baseline, we used the RealSense SDK's default hole-filling algorithm, a simple local image filter which locally replaces invalid pixels with the nearest valid pixel to the left. We evaluate all methods both quantitatively and visually.

5.2.1 Quantitative Analysis. To assess the quality, we use two quality metrics: PSNR and RMSE, as described in Section 5.1. The results are shown in Table 6 and we have the following observations from the results.
① On real-world datasets, *VoxDepth* achieves an average PSNR improvement of 31% compared to the best-performing competing method in each dataset.

Table 6. Comparison of the quality of different methods based on PSNR (dB, higher is better) and M-RMSE (lower is better). The numbers in bold indicate the best values. Throughout this paper, the best and second-best results of each test setting are highlighted in bold red and underlined blue, respectively.

Quality Metric	**Method Abbr.**	**VoxDepth** *VD*	**RealSense** *RS [19]*	**GSmooth** *GS [22]*	**DeepSmooth** *DS [27]*
PSNR (dB) ↑	LN	**17.46**	13.04	13.14	14.16
	MB	**17.10**	13.17	15.36	10.51
	KTS	17.69	14.60	18.59	**22.87**
	KTC	**20.32**	17.66	18.59	14.89
	PLF	**15.81**	13.55	12.97	13.62
	PLW	**16.05**	13.73	13.14	13.37
Masked RMSE ↓	LN	**112.25**	167.25	179.08	180.03
	MB	177.61	**175.61**	176.75	178.04
	KTS	**102.23**	139.69	172.02	175.68
	KTC	**126.15**	144.70	177.93	178.89
	PLF	**157.95**	159.15	178.70	178.38
	PLW	169.79	**164.79**	177.59	178.87

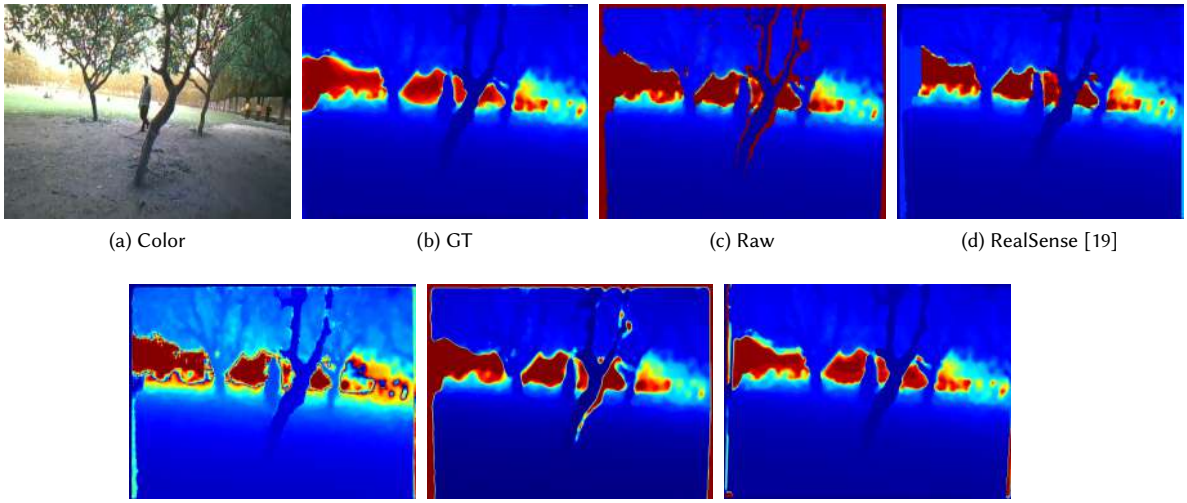
② On synthetic datasets, *VoxDepth* outperforms other methods with an average PSNR gain of 14.07%, with the exception of the Kite Sunny (KTS) dataset, where DeepSmooth performs better. This indicates that *VoxDepth* exhibits stronger generalization capability across varied data distributions.
③ In terms of RMSE, *VoxDepth* shows an average improvement of 25.23% over the next-best method on most datasets, except for the MB and PLW benchmarks.
④ Even in these exceptions, the performance gap is marginal, with *VoxDepth* trailing by only 2.08%, further supporting its robustness.

5.2.2 Visual Analysis. In Fig. 11, we see that *VoxDepth* surpasses all other techniques in terms of the accuracy of the estimated depth, mostly due to our emphasis on rectifying erroneous patches caused by algorithmic noise such as occlusion. From the figure, it is clear that *VoxDepth* has superior occlusion hole filling capabilities. However, all the methods are able to correct flickering noise in the depth images.

5.3 Runtime Performance Analysis

In this section, we compare the runtime performance of *VoxDepth* with state-of-the-art depth rectification algorithms. To make the comparison fair, we implement the most efficient version of all the state-of-the-art methods. For example, to ensure fast inference in DeepSmooth, we adopt the architecture recommended by the authors, incorporating a gated depth encoder [53] and an EfficientNet-lite-based [2] color encoder. Moreover, we conducted extensive tests to identify the most latency-efficient configuration of DeepSmooth with minimal quality trade-off. The model was converted to ONNX [3] and executed using CUDA and TensorRT providers to optimize runtime. As DeepSmooth is implemented in Python using PyTorch, we report only its pure inference time for fairness. We re-implemented it in C++ with custom CUDA kernels to reduce latency on the edge device.

5.3.1 Frame Rate. The first key metric that we evaluate is the frame rate (should be $\geq$ 20 FPS). The results are shown in Table 7 and the observations from the results are as follows:
①*VoxDepth* provides frame rates that exceed our requirements. It is strongly competitive with alternative approaches. A frame rate of 26.7 FPS can be provided, which meets our target.



(a) Color (b) GT (c) Raw (d) RealSense [19]

(e) DeepSmooth [27] (f) GSmooth [22] (g) VoxDepth

Fig. 11. Visual comparisons against state-of-the-art methods.

② The pipelined approach exhibits a 21% speedup as compared to the non-pipelined implementation on the Jetson Nano device.

Table 7. Frame rates of different methods on the Jetson Nano board

Metric	FPS
VD	26.71
VD- (Nopipe)	21.91
RS [19]	**39.52**
GS [22]	16.89
DS [27]	2
DS-fast [27]	12.23
Swarm [23]	14

Table 8. Latency (ms) of each step

Step	Jetson Nano	PC
Fusion	280	85
Inpainting	94	2
Transform	6	4
Combine	13	3

Table 9. Memory Consumption of each step

Step	Memory (in MB)
Fusion	128
Inpainting	1.2
Transform	3.4
Combine	4.5

5.3.2 Comparison with Lightweight Neural Networks. While DeepSmooth's architecture was not originally designed for embedded systems, we employ multiple optimization strategies to make it faster and suitable for embedded systems using reduced/distilled neural networks. To reduce the network latency, we perform network pruning using a state-of-the-art algorithm, NetAdapt [31] (similar to previous depth estimation work [49]). NetAdapt automatically and iteratively identifies and removes redundant channels from the feature maps to reduce the computational complexity. Furthermore, depthwise separable convolutional layers used in the DeepSmooth are not yet fully optimized for fast runtime. This motivates the need for hardware-specific compilation. We use the TVM compiler stack [13] to compile the network for deployment on embedded platforms such as Jetson Nano. From Table 7, we can see that despite the extensive optimizations in DeepSmooth, it (DS-fast) achieves only 12.23 FPS, which remains significantly below the target frame rate (20 FPS) for real-time embedded applications.

5.3.3 Latency. Each of the components described in Section 4 had to adhere to strict time constraints. Otherwise, it would not have been possible to achieve a frame rate of 26.7 FPS. We list the average latencies for each of these components in Table 8. The first row shows the numbers for Jetson Nano. For the purpose of comparison and sanity checking, we present the results for a workstation PC as well. As expected, we observe that the fusion step takes the longest time (2.9× slower than the next slowest task that is inpainting). This can be explained by the fact that in the fusion step, visual odometry is used to estimate the camera motion in 3D space, and multiple point clouds are fused together to create a dense scene representation. Inpainting also takes a significant amount of time because it performs many operations per pixel (maximum in a window). Note that the fusion and inpainting tasks are performed only once (at the beginning) of an epoch.

5.3.4 Memory Consumption. As mentioned in Section 1, along with the latency there is a constraint on the memory overhead (< 4 GB) for depth rectification in real life. We need to ensure that every part of our system uses memory efficiently so it can run on embedded devices like the NVIDIA Jetson Nano. Table 9 shows how much memory each stage of the system uses, including the voxel-based fusion step. As expected, the 3D fusion step has the largest memory consumption. However, the total memory usage of *VoxDepth* remains well within the 4 GB limit, demonstrating its suitability for resource-constrained hardware.

5.3.5 Power Consumption. We use the *jtop* [9] tool to get the power usage on the Jetson board. Fig. 12 shows a comparison of various depth correction methods in terms of their average power consumption. We find that ❶ *VoxDepth* has 2.9% lower power consumption than the closest competing method GSmooth [22] and ❷ 28.8% lower power consumption than the ML-based method DeepSmooth [27]. Because it does not use a heavy neural network for inference, it is more power-efficient than DeepSmooth. The reason for lower power consumption than GSmooth is because *VoxDepth* spends far less power in memory operations. Because we use the same 2D template throughout the epoch, we need to store very little information as compared to competing proposals that rely on much larger stores of information and frequent computation.

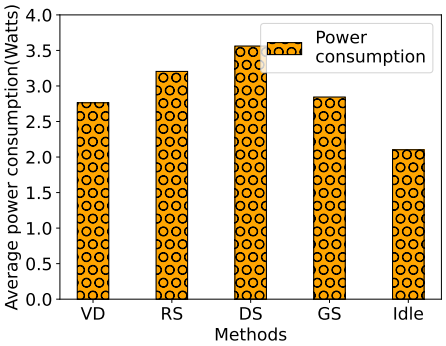


Fig. 12. Power consumption of the different techniques

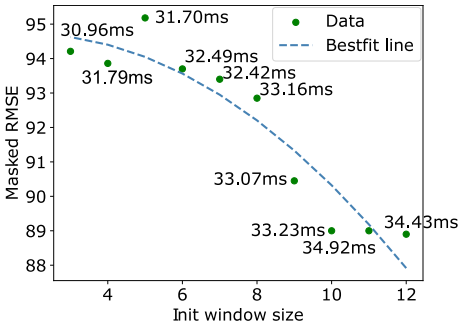


Fig. 13. Depth image quality vs voxel fusion window size. The number against each data point shows the average time taken to generate a single frame in ms.

5.4 Ablation Study

To thoroughly understand the impact of different components in *VoxDepth*, we conducted an ablation study focusing on four key elements: point cloud fusion, inpainting, algorithmic noise correction, and flickering noise filter. This analysis helps to isolate the contribution of each component to the overall performance of *VoxDepth*. We evaluate four distinct variants of *VoxDepth*: one where 3D point cloud fusion is replaced with 2D fusion, another with the sparse fused point cloud (no inpainting), a third without template image registration for mitigating the algorithmic noise, and a forth without the median filter that corrects the flickering noise.. The quantitative comparison of these variants along with the original method is given in Table 10. We make the following observations from the results:
① The absence of the 3D point cloud fusion module results in the most significant degradation in PSNR and an increase in RMSE. This is expected, as 2D images primarily encode visual appearance, while 3D point clouds preserve accurate geometric structure essential for consistent depth estimation.
② Excluding the image registration component also results in a notable performance decline, particularly due to its role in correcting algorithmic holes.
③ Finally, removing the median filtering step affects quality to a lesser extent, as flickering noise is more localized and less disruptive than geometric artifacts.

Table 10. Quantitative comparison of various variants of *VoxDepth* in terms of PSNR (dB), SSIM, and LPIPS. *w/o 3DF* refers to without 3D fusion. *w/o IP* refers to the case where inpainting is not used. *w/o IR* refers to without template image registration taken into account. *w/o MF* refers to without using median filtering.

Quality Metric	Method Abbr.	VoxDepth	w/o 3DF	w/o IP	w/o IR	w/o MF
PSNR (dB) ↑	LN	**17.46**	10.02	15.28	11.25	16.95
	MB	**17.10**	10.72	14.63	10.36	16.32
	KTS	**17.69**	11.36	16.31	10.99	16.11
	KTC	**20.32**	13.01	18.69	13.58	19.75
	PLF	**15.81**	09.89	13.29	10.35	14.20
	PLW	**16.05**	11.22	15.98	10.36	15.03
Masked RMSE ↓	LN	**112.25**	178.33	120.36	169.36	119.23
	MB	**177.61**	197.36	180.25	193.23	179.23
	KTS	**102.23**	165.31	110.32	158.36	113.23
	KTC	**126.15**	159.25	129.98	160.34	128.79
	PLF	**157.95**	188.36	161.25	179.36	161.36
	PLW	**169.79**	193.25	173.36	199.25	176.02

5.5 Sensitivity Analysis

5.5.1 Fusion Window Size. To understand what the ideal number of frames should be in the fusion window, we plot the quality of depth images for different fusion window sizes in Fig. 13. We also annotate the average time taken to generate a single frame beside each data point. We find that there is a clear trade-off between quality and latency. In our experiments, we use a frame window of size 10 for voxel fusion. This is the most optimal configuration. ① A 10-frame fusion window, when compared to the largest window size of 12, generates similar quality depth images but reduces latency by 3.61%.

5.5.2 Template Replacement Criteria. We conduct a sensitivity analysis to study how the template replacement thresholds impact the overall performance of our system. As described in Section 4.4, two parameters govern when the

system transitions to a new epoch: ① the feature distance threshold and ② the minimum number of good matches. These thresholds assess the similarity between the current 2D template and the incoming depth frame. Figures 14 and 15 illustrate how varying these thresholds affects the masked RMSE of the output depth images. Each data point is annotated with the corresponding number of state (epoch) transitions, which directly correlates with the frequency of point cloud fusion. Since fusion is the most computationally expensive step, frequent epoch switches lead to higher latency and lower frame rates. Our analysis reveals a clear trade-off: stricter thresholds improve depth quality but increase the number of fusion operations, whereas relaxed thresholds improve runtime efficiency at the cost of accuracy. Based on this study, we selected a feature distance threshold of 20 and a good match count threshold of 5 as the optimal configuration, balancing accuracy and computational cost.

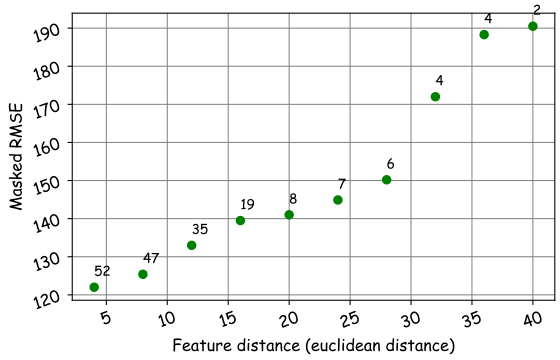


Fig. 14. Depth image quality vs feature distance. The number against each data point shows the number of state switches.

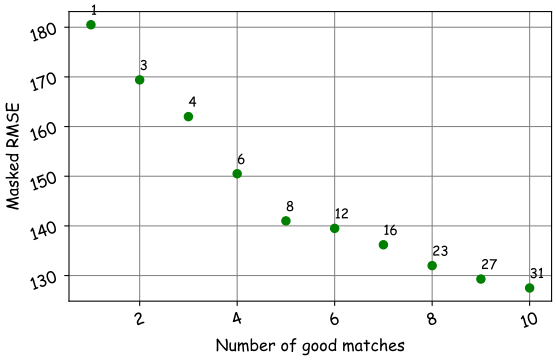


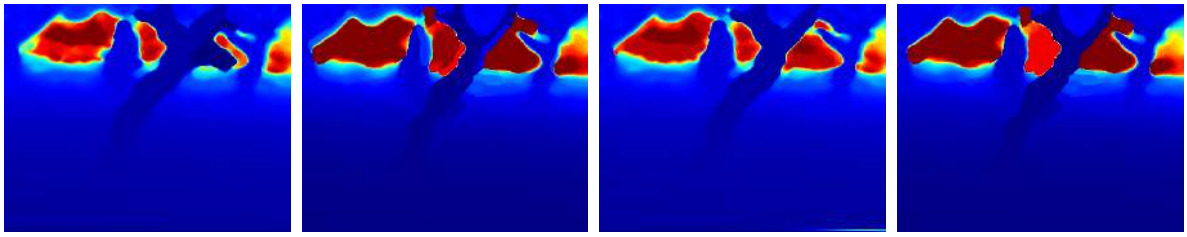
Fig. 15. Depth image quality vs number of good matches. The number against each data point shows the number of state switches.

5.5.3 State Switch: Optimal Time and Frequency. We measured the average pixel movement between frames by generating an optical flow (refer to Section 3.5). We found that RGB images 50 frames apart showed an average pixel movement of 6.75 pixels and the quality of registration (see Section 5.5.5) was 16.06 dB. When we repeated this experiment with images 150 frames apart, the average pixel movement went up to 10.08 pixels, and the quality of registration went down to 12.61 decibels. The conclusion here is that there is a strong relationship between these three variables (positive or negative): the distance between frames, the average pixel movement and the image registration quality.

The frequency of switching between fusion and correction states (shown in Fig. 5c) depends on the dataset in use, the type of motion in the dataset and the amount of textural information in the scene. With all other parameters constant, the number of state switches recorded for each dataset is presented in Table 11. In most cases, the synthetically generated datasets require fewer switches due to consistent lighting and contrast values across frames. Except the Kite Sunny dataset, we observed a 40.33% lower number of state switches in synthetically generated datasets as compared to datasets generated using the RealSense camera. One reason behind the high frequency of switching would be fast changes in the scene that the camera observes. Such a scenario is unlikely to happen in the case of household robots or autonomous drones. The insights in this study led us to the algorithm that decides when to end an epoch and start a new one (state switch).

Table 11. Number of state switches for each dataset while processing 500 frames

LN	MB	KTS	KTC	PLF	PLW
10	10	4	14	4	5



(a) Bilinear interpolation (b) Convex upsampling (c) Grayscale dilation (d) Ground truth

Fig. 16. Visual comparison of inpainting methods on depth images.

5.5.4 Impact of Inpainting Methods. The two dimensional projection of a point cloud is a sparse depth image as shown in Fig. 7a. To create a noise-free dense scene representation that contains more information, the sparse projection would need to be inpainted. Conventional image processing methods such as bilinear interpolation are fast but fail to maintain the sharpness of edges. In certain situations, they may also create unusual visual defects as illustrated in Fig. 16a. Contemporary inpainting methods that use learning such as convex upsampling [17], yield nearly flawless outcomes (see Fig. 16b). However, they are prohibitively slow.

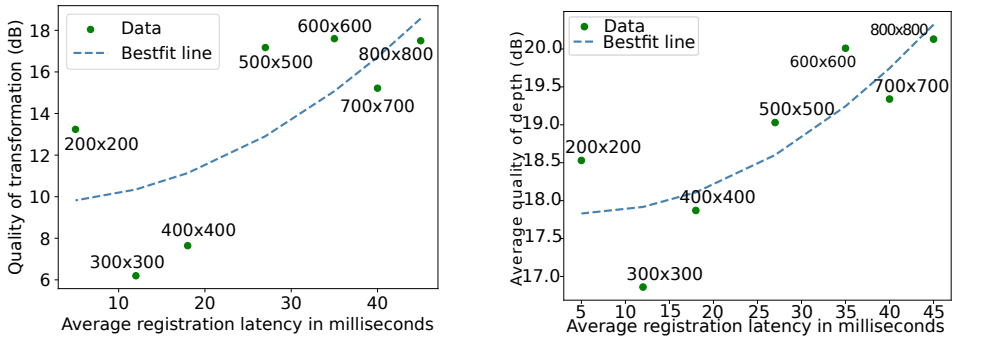
We implemented three methods for the purpose of comparison: bilinear interpolation[38], the convex upsampling method proposed by Teed et al. [43] and grayscale dilation [47]. Grayscale dilation is a local image filter that is used for noise reduction in image processing tasks where preserving structural information is important. The quality of generated images and associated latencies on the Jetson Nano board are shown in Table 12. We find that the latency associated with grayscale dilation falls within the acceptable threshold for producing corrected frames at 20 FPS; it performs better than bilinear interpolation.

Table 12. Comparison of the image quality and latencies of different inpainting methods

Method	**PSNR** (dB)	**Latency** (ms)
Bilinear interpolation	16.35	8
Convex upsampling	34.39	237
Grayscale dilation	17.46	94

5.5.5 Impact of the Frame Size on the Registration Latency and Quality. To speed up the process of image registration, we can reduce the dimensions of the image. This way, the matching algorithm can find matching pixels in frames faster since the search area reduces with a reduction in image dimensions. This comes at the cost of a lower accuracy of the image registration process. In Fig. 17, we show this relationship graphically. We measure the quality of registration as the PSNR of the transformed frame vis-a-vis the original frame. We also plot the average quality of the corrected depth images against the respective latencies associated with performing registration on the resized images. We observe a trade-off between the image quality and the latency that broadly shows a monotonically increasing trend with the frame size. Occasional deviation from a monotonically increasing trend can be attributed to image-specific variations,

noise induced due to the data itself and camera motion. The figure suggests that the best frame size is 200×200 (good quality with low latency). This is what we choose for our experiments. One may always argue that this is an artifact of our experiment and datasets. We tested with all kinds of images, and the broad conclusion is that a smaller resolution is good enough from a quality perspective given that preservation of low-frequency features tends to affect image registration the most (also observed in[45]). Needless to say, smaller frame sizes are always desirable from a latency perspective. Hence, even with other datasets the conclusion is not expected to be very different.



(a) Quality of registration vs frame latency

(b) Average quality of the corrected depth images vs frame latency

Fig. 17. Quality of registration vs latency. The annotation against each data-point depicts the dimensions of the resized frame.

5.6 Impact of Noisy Depth on 3D Perception Related Applications

Unrectified depth images can introduce significant geometric inconsistencies that negatively affect a wide range of 3D perception tasks [51]. For example: in 3D reconstruction, they lead to ghosting and surface distortion; in SLAM, they cause drift and inaccurate pose estimation. Moreover, tasks like 3D object detection suffer from blurred object boundaries. Now, to measure the quantitative impact of unrectified depth images, we conduct experiments on two representative 3D perception applications: (i) drone swarming, where accurate depth is critical for collision avoidance, and (ii) RGB-D semantic segmentation, where depth quality directly affects segmentation accuracy. Below, we describe these experiments in detail.

5.6.1 Drone Swarming. In order to signify the importance of rectified depth images in drone swarming, we introduced distorted depth data into a drone swarming simulator written using the Unity engine (SmrtSwarm [8]). We used a city skyline scene with a leader drone and seven drones following the leader in a swarming formation. Each drone is equipped with a simulated *stereo camera* (depth estimator) to measure its distance from obstacles and other drones. The collision ratio is defined as the proportion of simulated runs that result in collisions divided by the total number of runs. We conducted these simulated runs by introducing varying levels of sensor noise to the simulated depth measurement camera. This noise is defined by a noise ratio threshold, θ. A noise ratio, β is uniformly sampled from the range $[-\theta, +\theta]$. The noise ratio is subsequently multiplied by the depth measurement and added to the depth in order to obtain the new depth value. θ in our experiments is a single precision floating point number varying from 0.3 to 1.0. In Equation 9, D_{true} is the depth measurement value from the simulated stereo camera. D_{noise} is the noise added to depth measurement.

$$D_{noise} = D_{true} + D_{true} \times \beta \tag{9}$$

We present our findings in Fig. 18. ❶ As noise increases, the collision frequency grows super-linearly (almost quadratically). ❷ After θ exceeds 0.5, increasing θ by 2× leads to an 8× increase in the number of collisions.

5.6.2 Semantic Segmentation. We further evaluate the effect of depth rectification on semantic segmentation performance using the latest state-of-the-art RGB-D segmentation model, **DFormerV2**, published at CVPR 2025 [52]. We use the best-performing pre-trained variant of the model for our comparison. We prepare two evaluation sets: ① RGB + *noisy* depth image (captured using Intel RealSense), ② RGB + *rectified* depth image (processed with our proposed *VoxDepth* rectification pipeline). For evaluation, we use a standard quality metric, mean Intersection over Union (mIoU), to measure the segmentation accuracy, averaged across all semantic categories. The results shown in Table 13 clearly demonstrate that rectified depth images significantly improve semantic segmentation accuracy, underscoring the value of depth correction even for high-performing modern models.

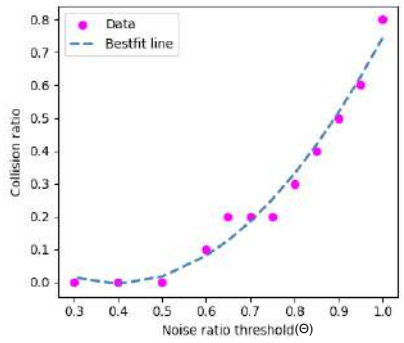


Fig. 18. Relation between the number of collisions in simulations and the depth measurement noise

Table 13. Impact of Noisy Depth on Semantic Segmentation

Quality Metric	Method Abbr.	DFormerv2-Noisy Depth	DFormerv2-Rectified Depth
mIoU	LN	**53.23**	53.04
	MB	**54.36**	53.17
	KTS	**51.24**	49.60
	KTC	**54.26**	53.28
	PLF	**55.81**	54.55
	PLW	**54.05**	53.73

6 CONCLUSION

The key premise of our paper is that existing ML methods are accurate yet slow, whereas non-ML techniques are quite fast but do not provide adequate quality. Given that many works in the edge computing domain have pointed out that a good frame rate (at least 20 FPS) is required, there was a strong need to develop such a solution that can run on embedded boards. Our proposal, *VoxDepth*, was able to successfully provide a frame rate of 27 FPS on an NVIDIA Jetson Nano board and also outperform the state of the art (both ML-based and non-ML) in terms of the depth image quality.

It particularly did very well in removing algorithmic noise (improvement in masked RMSE by 25%) and PSNR (31% better). It also proved to be 58% faster that the closest competing proposal in the literature. The key learnings from the paper are as follows:

(1) To rectify 2D depth images, it is a good idea to maintain a stable 3D representation of the scene in the form of a point cloud. It preserves important 3D information.
(2) Instead of relying on a lot of ephemeral data, it is a better idea to split time into epochs and use a single fused point cloud as the basis for scene rectification throughout an epoch. It provides a stable baseline.
(3) Converting the fused point cloud to a template image has two key advantages. First it allows us to use standard image registration techniques, and second, it is very performance efficient (faster than 3D↔2D comparison and rectification).
(4) Algorithmic holes are an important source of noise and are fundamentally different from random flickering noise. They represent the systematic component of noise. This work rightly takes cognizance of them and also uses metrics like the masked RMSE metric to specifically assess whether they have been properly filled or not.
(5) Using a pipelined approach is a wise idea in a heterogeneous system that comprises CPUs, GPUs and accelerators. It ensures that all the parts of the system are used simultaneously and there is no idling.

ACKNOWLEDGMENTS

This research was supported by the Ministry of Electronics and Information Technology (MeitY), Government of India (Grant #3095895).

REFERENCES

[1] [n. d.]. https://www.nvidia.com/en-in/autonomous-machines/embedded-systems/jetson-orin/ [Online; accessed 2024-06-11].
[2] [n. d.]. Higher accuracy on vision models with EfficientNet-Lite. https://blog.tensorflow.org/2020/03/higher-accuracy-on-vision-models-with-efficientnet-lite.html [Online; accessed 2024-03-30].
[3] [n. d.]. ONNX | Home. https://onnx.ai/ [Online; accessed 2024-06-11].
[4] [n. d.]. Tegra Linux Driver. https://docs.nvidia.com/jetson/archives/l4t-archived/l4t-325/index.html [Online; accessed 2024-03-30].
[5] Eduardo Assunção, Pedro D Gaspar, Ricardo Mesquita, Maria P Simões, Khadijeh Alibabaei, André Veiros, and Hugo Proença. 2022. Real-time weed control application using a jetson nano edge device and a spray mechanism. *Remote Sensing* 14, 17 (2022), 4217.
[6] Razmik Avetisyan, Christian Rosenke, Martin Luboschik, and Oliver Staadt. 2016. Temporal filtering of depth images using optical flow. (2016).
[7] Herbert Bay, Tinne Tuytelaars, and Luc Van Gool. 2006. Surf: Speeded up robust features. In *Computer Vision–ECCV 2006: 9th European Conference on Computer Vision, Graz, Austria, May 7-13, 2006. Proceedings, Part I 9*. Springer, 404–417.
[8] Nikita Bhamu, Harshit Verma, Akanksha Dixit, Barbara Bollard, and Smruti R Sarangi. 2023. SmrtSwarm: A Novel Swarming Model for Real-World Environments. *Drones* 7, 9 (2023), 573.
[9] Raffaello Bonghi. [n. d.]. Jetson-stats. https://rnext.it/jetson_stats/ [Online; accessed 2024-03-30].
[10] Antoni Buades, Bartomeu Coll, and J-M Morel. 2005. A non-local algorithm for image denoising. In *2005 IEEE CVPR*, Vol. 2. Ieee, 60–65.
[11] Michael Calonder, Vincent Lepetit, Christoph Strecha, and Pascal Fua. 2010. Brief: Binary robust independent elementary features. In *Computer Vision–ECCV 2010*. Springer, 778–792.
[12] Anh-Quan Cao and Raoul de Charette. 2023. Scenerf: Self-supervised monocular 3d scene reconstruction with radiance fields. In *Proceedings of CVPR*. 9387–9398.
[13] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. 2018. {TVM}: An automated {End-to-End} optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 578–594.
[14] Yun Chen, Bin Yang, Ming Liang, and Raquel Urtasun. 2019. Learning joint 2d-3d representations for depth completion. In *Proceedings of CVPR*. 10023–10032.
[15] Weichen Dai, Yu Zhang, Ping Li, Zheng Fang, and Sebastian Scherer. 2020. Rgb-d slam in dynamic environments using point correlations. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 44, 1 (2020), 373–389.
[16] Jeffrey Delmerico, Titus Cieslewski, Henri Rebecq, Matthias Faessler, and Davide Scaramuzza. 2019. Are we ready for autonomous drone racing? the UZH-FPV drone racing dataset. In *2019 International Conference on Robotics and Automation (ICRA)*. IEEE, 6713–6719.

[17] David Ferstl, Christian Reinbacher, Rene Ranftl, Matthias Rüther, and Horst Bischof. 2013. Image guided depth upsampling using anisotropic total generalized variation. In *Proceedings of the IEEE international conference on computer vision*. 993–1000.

[18] Michael Fonder and Marc Van Droogenbroeck. 2019. Mid-air: A multi-modal dataset for extremely low altitude drone flights. In *Proceedings of CVPR workshops*. 0–0.

[19] Anders Grunnet-Jepsen and Dave Tong. 2018. Depth post-processing for intel® realsense™ d400 depth cameras. *New Technologies Group, Intel Corporation* 3 (2018).

[20] Mostafa Mahmoud Ibrahim, Qiong Liu, Rizwan Khan, Jingyu Yang, Ehsan Adeli, and You Yang. 2020. Depth map artefacts reduction: A review. *IET Image Processing* 14, 12 (2020), 2630–2644.

[21] Saif Imran, Xiaoming Liu, and Daniel Morris. 2021. Depth Completion With Twin Surface Extrapolation at Occlusion Boundaries. In *Proceedings of CVPR (CVPR)*. 2583–2592.

[22] Abm Tariqul Islam, Martin Luboschik, Anton Jirka, and Oliver Staadt. 2018. gSMOOTH: A Gradient based Spatial and Temporal Method of Depth Image Enhancement. In *Proceedings of Computer Graphics International 2018*. 175–184.

[23] Kader Monhamady Kabore and Samet Güler. 2021. Distributed formation control of drones with onboard perception. *IEEE/ASME Transactions on Mechatronics* 27, 5 (2021), 3121–3131.

[24] Leonid Keselman, John Iselin Woodfill, Anders Grunnet-Jepsen, and Achintya Bhowmik. 2017. Intel realsense stereoscopic depth cameras. In *Proceedings of the IEEE CVPR workshops*. 1–10.

[25] Jin-Hyeok Kim, Tae-Hui Lee, Yamin Han, and Heejung Byun. 2021. A study on the design and implementation of multi-disaster drone system using deep learning-based object recognition and optimal path planning. *KIPS Transactions on Computer and Communication Systems* 10, 4 (2021), 117–122.

[26] Johannes Kopf, Michael F Cohen, Dani Lischinski, and Matt Uyttendaele. 2007. Joint bilateral upsampling. *ACM Transactions on Graphics (ToG)* 26, 3 (2007), 96–es.

[27] Sriram Krishna and Basavaraja Shanthappa Vandrotti. 2023. DeepSmooth: Efficient and Smooth Depth Completion. In *Proceedings of CVPR*. 3357–3366.

[28] RA Lane and NA Thacker. 1994. Stereo vision research: An algorithm survey. *URL citeseer. ist. psu. edu/lane96stereo. html* (1994).

[29] Przemyslaw A Lasota, Gregory F Rossano, and Julie A Shah. 2014. Toward safe close-proximity human-robot interaction with standard industrial robots. In *2014 IEEE CASE*. IEEE, 339–344.

[30] David Luebke. 2008. CUDA: Scalable parallel programming for high-performance scientific computing. In *2008 5th IEEE international symposium on biomedical imaging: from nano to macro*. IEEE, 836–838.

[31] Raúl Marichal, Ernesto Dufrechou, and Pablo Ezzatti. 2024. Avoiding Training in the Platform-Aware Optimization Process for Faster DNN Latency Reduction. In *2024 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 311–320.

[32] Sergey Matyunin, Dmitriy Vatolin, Yury Berdnikov, and Maxim Smirnov. 2011. Temporal filtering for depth maps generated by kinect depth camera. In *2011 3DTV Conference: The True Vision-Capture, Transmission and Display of 3D Video (3DTV-CON)*. IEEE, 1–4.

[33] Yoshiki Mizukami and Katsumi Tadamura. 2007. Optical flow computation on compute unified device architecture. In *14th International Conference on Image Analysis and Processing (ICIAP 2007)*. IEEE, 179–184.

[34] MordorIntelligence. [n. d.]. Autonomous Mobile Robots Market. https://www.mordorintelligence.com/industry-reports/autonomous-mobile-robot-market. [Online; accessed 2024-01-20].

[35] Marius Muja and David Lowe. 2009. Flann-fast library for approximate nearest neighbors user manual. *Computer Science Department, University of British Columbia, Vancouver, BC, Canada* 5, 6 (2009).

[36] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. 2015. U-net: Convolutional networks for biomedical image segmentation. In *Medical Image Computing and Computer-Assisted Intervention–MICCAI 2015*. Springer, 234–241.

[37] Ethan Rublee, Vincent Rabaud, Kurt Konolige, and Gary Bradski. 2011. ORB: An efficient alternative to SIFT or SURF. In *2011 International conference on computer vision*. Ieee, 2564–2571.

[38] Yang Sa. 2014. Improved bilinear interpolation method for image fast processing. In *2014 7th International Conference on Intelligent Computation Technology and Automation*. IEEE, 308–311.

[39] Dmitry Senushkin, Mikhail Romanov, Ilia Belikov, Nikolay Patakin, and Anton Konushin. 2021. Decoder modulation for indoor depth completion. In *2021 IEEE/RSJ IROS*. IEEE, 2181–2188.

[40] Shuran Song and Jianxiong Xiao. 2014. Sliding shapes for 3d object detection in depth images. In *Computer Vision–ECCV 2014: 13th European Conference*. Springer, 634–651.

[41] Frank Steinbrücker, Jürgen Sturm, and Daniel Cremers. 2011. Real-time visual odometry from dense RGB-D images. In *2011 IEEE ICCV Workshops*. IEEE, 719–722.

[42] Vladimiros Sterzentsenko, Leonidas Saroglou, Anargyros Chatzitofis, Spyridon Thermos, Nikolaos Zioulis, Alexandros Doumanoglou, Dimitrios Zarpalas, and Petros Daras. 2019. Self-supervised deep depth denoising. In *Proceedings of CVPR*. 1242–1251.

[43] Zachary Teed and Jia Deng. 2020. Raft: Recurrent all-pairs field transforms for optical flow. In *Computer Vision–ECCV 2020: 16th European Conference, Glasgow, UK, August 23–28, 2020, Proceedings, Part II 16*. Springer, 402–419.

[44] Trisha Thadani, Faiz Siddiqui, Rachel Lerman, and Jeremy B. Merrill. 2023. Tesla drivers run Autopilot where it's not intended — with deadly consequences. https://www.washingtonpost.com/technology/2023/12/10/tesla-autopilot-crash/ [Online; accessed 2024-03-30].

[45] Georgios Tzimiropoulos, Vasileios Argyriou, Stefanos Zafeiriou, and Tania Stathaki. 2010. Robust FFT-based scale-invariant image registration with image gradients. *IEEE transactions on pattern analysis and machine intelligence* 32, 10 (2010), 1899–1906.

[46] Uraizee, Trisha Thadani Rachel Lerman, Imogen Piper, Faiz Siddiqui, and Irfan. 2023. Inside the final seconds of a deadly Tesla Autopilot crash. https://www.washingtonpost.com/technology/interactive/2023/tesla-autopilot-crash-analysis/ [Online; accessed 2024-03-30].

[47] Luc Vincent. 1993. Morphological grayscale reconstruction in image analysis: applications and efficient algorithms. *IEEE transactions on image processing* 2, 2 (1993), 176–201.

[48] Deepak Geetha Viswanathan. 2009. Features from accelerated segment test (fast). In *Proceedings of the 10th workshop on image analysis for multimedia interactive services, London, UK*. 6–8.

[49] Diana Wofk, Fangchang Ma, Tien-Ju Yang, Sertac Karaman, and Vivienne Sze. 2019. Fastdepth: Fast monocular depth estimation on embedded systems. In *2019 International Conference on Robotics and Automation (ICRA)*. IEEE, 6101–6108.

[50] Yan Xu, Xinge Zhu, Jianping Shi, Guofeng Zhang, Hujun Bao, and Hongsheng Li. 2019. Depth completion from sparse lidar data with depth-normal constraints. In *Proceedings of CVPR*. 2811–2820.

[51] Chenggang Yan, Zhisheng Li, Yongbing Zhang, Yutao Liu, Xiangyang Ji, and Yongdong Zhang. 2020. Depth Image Denoising Using Nuclear Norm and Learning Graph Model. *ACM Trans. Multimedia Comput. Commun. Appl.* 16, 4, Article 122 (Dec. 2020), 17 pages. https://doi.org/10.1145/3404374

[52] Bo-Wen Yin, Jiao-Long Cao, Ming-Ming Cheng, and Qibin Hou. 2025. DFormerv2: Geometry Self-Attention for RGBD Semantic Segmentation. In *CVPR*.

[53] Jiahui Yu, Zhe Lin, Jimei Yang, Xiaohui Shen, Xin Lu, and Thomas S Huang. 2019. Free-form image inpainting with gated convolution. In *Proceedings of the IEEE/CVF international conference on computer vision*. 4471–4480.

[54] Yinda Zhang and Thomas Funkhouser. 2018. Deep depth completion of a single rgb-d image. In *Proceedings of CVPR*. 175–185.