An Efficient Technique for Real-time Transformation of 2D to 3D Images With GPU Using CUDA Programming

by

Sadat Mahmud
22301301
Md. Rana Mustafa
21101060
Ananda Mitra
21101268
Sajjad Hossain Shanto
20201010
Mohammad Nazibul Bashar Chowdhury
21301736

A thesis submitted to the Department of Computer Science and Engineering
in partial fulfillment of the requirements for the degree of
B.Sc. in Computer Science

Department of Computer Science and Engineering
BRAC University
February 2026

Declaration

It is hereby declared that

1. The thesis submitted is my/our own original work while completing degree at Brac University.
2. The thesis does not contain material previously published or written by a third party, except where this is appropriately cited through full and accurate referencing.
3. The thesis does not contain material which has been accepted, or submitted, for any other degree or diploma at a university or other institution.
4. We have acknowledged all main sources of help.

Student's Full Name & Signature:

Sadat Mahmud
22301301

Md. Rana Mustafa
21101060

Ananda Mitra
21101268

Sajjad Hossain Shanto
20201010

Mohammad Nazibul Bashar Chowdhury
21301736

Approval

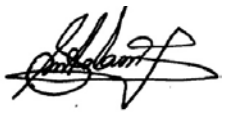
The thesis/project titled "An efficient technique for real-time transformation of 2D to 3D images using CUDA programming" submitted by

1. Sadat Mahmud (22301301)
2. Md. Rana Mustafa (21101060)
3. Ananda Mitra (21101268)
4. Sajjad Hossain Shanto (20201010)
5. Mohammad Nazibul Bashar Chowdhury (21301736)

Of Fall, 2026 has been accepted as satisfactory in partial fulfillment of the requirement for the degree of B.Sc. in Computer Science on February 24, 2026.

Examining Committee:

Supervisor:
(Member)



Dr. Md. Ashraful Alam
Associate Professor
Department of Computer Science and Engineering
Brac University

Program Coordinator:
(Member)

Md. Golam Rabiul Alam, PhD
Professor
Department of Computer Science
and Engineering
Brac University

Head of Department:
(Chair)

Dr. Sadia Hamid Kazi
Associate Professor & Chairperson
Department of Computer Science
and Engineering
Brac University

Acknowledgement

Firstly, all praise to the Great Allah for whom our thesis have been completed without any major interruption.
Secondly, to our supervisor Dr. Md. Ashraful Alam sir for his kind support and advice in our work. He helped us whenever we needed help.
And finally to our parents without their throughout sup-port it may not be possible. With their kind support and prayer we are now on the verge of our graduation.

Abstract

This thesis presents a complete 2D to 3D reconstruction system designed to run reliably on a low computational powered PC, where GPU memory, host memory, and disk bandwidth impose strict constraints. The pipeline begins with large-scale synthetic data generation from ShapeNet models, producing aligned RGB and depth observations for supervised learning. A ResUNet18 based monocular depth network is trained in LibTorch using a mask-aware objective to promote numerical stability and reduce invalid-depth regions in the predicted maps. To ensure continuous training without data starvation under limited resources, the system is implemented a producer consumer scheduling system design: a producer renders and stages batches to fast local storage, consumers stream and pre-process shards into the training loop, and a destroyer reclaims storage deterministically once a batch is fully consumed. This design bounds disk usage, prevents host RAM accumulation, and decouples rendering from training so the GPU remains saturated even when CPU-side work fluctuates. After inference, predicted camera-centric depth is lifted into explicit 3D geometry using CUDA-accelerated reconstruction, enabling dense point cloud and grid-mesh generation at image resolution with minimal overhead. The system is evaluated using runtime traces (GPU utilization, GPU/host memory, and CPU load) alongside standard depth estimation metrics aggregated across training batches, demonstrating sustained execution, stable memory behavior, and reconstruction-ready depth quality on resource-constrained hardware.

Keywords: 2D-to-3D reconstruction; monocular depth estimation; ResUNet; ResNet18; LibTorch; CUDA; GPU acceleration; producer–consumer; depth-to-point cloud; mesh generation; ShapeNet

Table of Contents

Chapter 1

Introduction

Reconstructing three-dimensional (3D) structure from two-dimensional (2D) image has been a reoccurring obstacle in computer vision because a conventional camera records a projection of the 3D world onto a 2D image plane, collapsing depth information in the process. This projection discards metric cues related to distance and spatial layout, meaning that multiple distinct 3D configurations can generate the same 2D observation. As a result, recovering 3D structure from a single image is inherently ambiguous and becomes an ill-posed inverse problem unless additional constraints, assumptions, or prior knowledge are introduced [4], [50]. Classical approaches address this ambiguity through multi-view geometry, where depth is inferred from correspondences across multiple images using techniques such as stereo, structure from motion, and multi-view reconstruction [4], [50]. While these methods can yield accurate reconstructions under calibrated and controlled acquisition conditions, they are less applicable in settings where only a single image is available or where capture conditions are unconstrained.

To understand the conversion of an image to a 3D object, we must firstly familiarize with what defines each. A 2D image is typically represented as a discrete grid of pixels indexed by (x, y). Each pixel stores three color values (R, G, B), often quantized to [0,255], which describe appearance but do not directly provide metric depth information. By contrast, a 3D representation introduces an additional spatial degree of depth, commonly expressed as (x, y, z), where the depth dimension z provides geometric context for spatial reasoning and reconstruction.

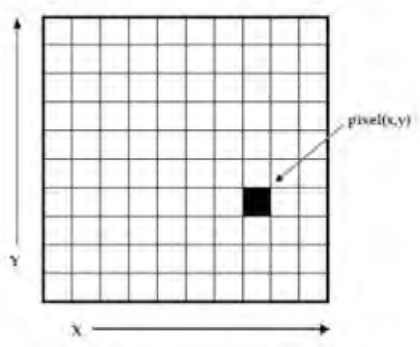


Figure 1.1: 2D image consists of an array of pixels, each pixel's location defined as coordinates (x, y).

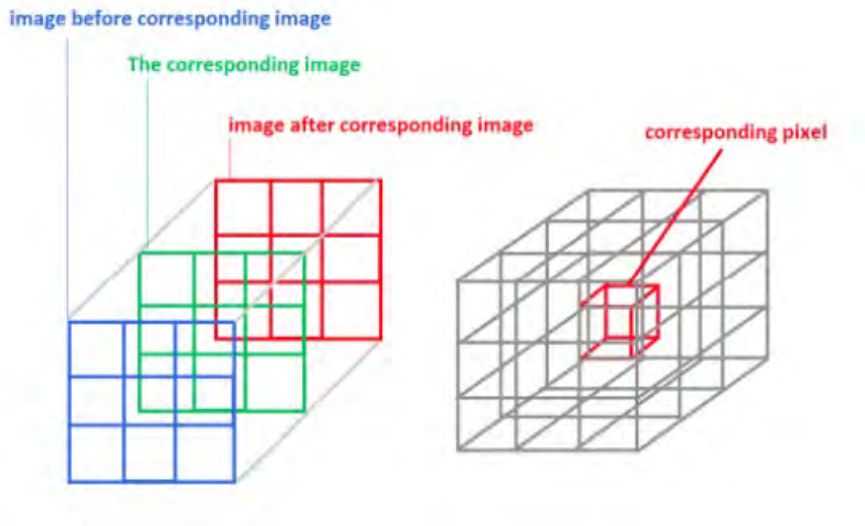


Figure 1.2: 3D image consists of an array of blocks, each pixel's location defined as coordinates (x, y, z).

At the same time, 2D-to-3D reconstruction is not only theoretically under-constrained but also computationally demanding. Traditional pipelines often involve sequential stages such as feature extraction, correspondence matching, depth inference, and geometric integration, where per-pixel operations and iterative optimization can become expensive as resolution and dataset size increase. Recent surveys and comparative studies emphasize that practical 2D-to-3D systems must balance reconstruction quality with computation, particularly when the target use case requires scalable processing rather than small, isolated experiments [55], [56]. In many applied settings, even modest increases in input resolution or dataset size can lead to disproportionate increases in runtime and memory usage, making system-level design a first-order concern rather than an implementation detail.

Learning-based methods have introduced a different route to 2D-to-3D reconstruction by predicting depth directly from monocular images. Convolutional neural networks (CNNs) can learn statistical cues correlated with depth—such as shading, texture gradients, object scale, and semantic context—and produce dense depth maps from a single input image [54], [56]. This shifts the reconstruction problem from explicit geometric reasoning to data-driven inference and expands applicability to single-view scenarios. However, learning-based depth estimation introduces new challenges: training requires significant computational and memory resources, and reconstruction from predicted depth still requires dense geometric processing. Consequently, the feasibility of building an end-to-end depth learning 2D-to-3D pipeline depends not only on model accuracy but also on whether training and reconstruction can be sustained efficiently under realistic hardware constraints [50], [56].

1.1 Problem Statement

The problem addressed in this thesis is the practical feasibility of performing 2D-to-3D reconstruction under realistic hardware constraints, particularly on modest hardware platforms. First, resource-limited machines often cannot train modern depth

estimation networks at scale. High-resolution inputs, extended training schedules, and memory-heavy optimization states place substantial pressure on GPU VRAM and host memory, which can make standard training workflows unstable or infeasible without careful pipeline design [50], [56]. Second, even when depth can be estimated, converting depth into usable 3D structure requires significant mathematical and geometric computation, including camera modeling, per-pixel back-projection, coordinate transformations, validity masking, and geometric integration into point clouds or meshes [4], [50]. Third, scalability is further constrained when training data must be generated through rendering and streamed to the learner. Rendering introduces CPU overhead and disk I/O pressure, and naive storage or loading strategies can exceed capacity or stall training due to data starvation unless dataset handling and synchronization are explicitly engineered [55], [56].

1.2 Research Objectives

This thesis pursues four objectives to address these constraints. First, it develops an complete 2D-to-3D pipeline that remains feasible on a low computational powered PC, supporting both monocular depth estimation and subsequent reconstruction. This includes selecting a network design and training procedure that can run under limited GPU memory while still producing depth maps suitable for downstream geometry generation [50], [56]. Second, it implements an SSD-backed producer consumer workflow that decouples rendering from training through sharded streaming. By structuring data generation, consumption, and cleanup as a bounded pipeline, the system aims to sustain GPU utilization, avoid data starvation, and keep disk and host memory usage within fixed limits even when training at scale [55], [56]. Third, it aims to accelerate 3D reconstruction using CUDA-enabled computation for dense back-projection and point/mesh generation, reducing CPU bottlenecks at image-grid scale and making reconstruction practical at higher resolutions [4], [50]. Finally, it evaluates both model quality and system behavior using standard depth metrics and runtime traces to verify that the overall pipeline remains stable and efficient during sustained operation [50], [56].

Research Questions

1. Can a complete 2D-to-3D pipeline run reliably on a low computational powered PC while keeping memory usage stable and training throughput consistent?

2. Does a hard drive backed producer consumer workflow effectively decouple rendering from training to prevent data starvation and keep disk and RAM usage bounded during large-scale training?

3. Does CUDA-accelerated 3D reconstruction reduce rendering time and CPU bottlenecks without degrading point cloud or mesh quality at practical image resolutions?

Chapter 2

Literature Review

2.1 Preliminaries

A 2D image is an observation of a 3D world, and the mapping from image measurements to 3D geometry is inherently constrained by camera projection [4], [50]. In many practical settings, the available input is a single RGB image or a limited set of views, while the desired output is a representation that supports downstream tasks such as visualization, surface extraction, measurement, or simulation [50]. This thesis focuses on 2D-to-3D reconstruction through a depth-centric approach, where reconstruction is driven by depth prediction and subsequent lifting into 3D [24], [26]. A depth-centric reconstruction pipeline is especially suitable for scalable systems because depth provides a dense geometric signal aligned with the image grid, enabling a direct interface between learned inference and geometric reconstruction [26], [50]. Once a depth map is available, it can be converted into 3D points through camera back-projection, and further processed into point clouds or meshes for downstream use [4], [50]. When multiple views exist, depth-derived geometry can be fused to improve density and reduce inconsistency [6], [7], [18]. This separation between depth estimation and geometric reconstruction is also useful in practice because it localizes learning complexity to depth prediction while keeping the reconstruction stage mathematically grounded in camera geometry [4], [50].
This chapter reviews the literature in two dimensions: (1) reconstruction approaches ranging from classical geometric methods to learning-based models for depth-centric 2D-to-3D conversion [4], [23], [24], [26], and (2) system-level architectures that support sustained throughput, including caching and pipeline parallelism for data movement and training/inference efficiency, as well as GPU acceleration through CUDA [25], [33], [38], [39]. Together, these define the methodological and engineering context for the proposed depth-centric 2D-to-3D reconstruction system.

2.1.1 2D-to-3D Reconstruction: Core Methods

The goal of 2D-to-3D reconstruction can be defined at multiple output levels. A common first-stage output is a depth map, which assigns a distance value to each pixel location (u, v) [4], [50]. From depth, a system can generate a 3D point cloud by projecting each pixel into 3D space using known or estimated camera intrinsics [4], [50]. Finally, a point cloud may be converted into a mesh (triangulated surface), which provides a continuous geometric representation that is often required

for rendering, simulation, or fabrication pipelines [50].
The general structure of a depth centric pipeline is the following: input image is processed and fed into a model for estimating depth - which outputs a predicted depth map [24], [26]. he depth map is then rectified using the camera parameters and converted into 3D points by using back projection [4], [50]. At this point, more filtering and consistency checks can be used for invalid regions, discontinuities or noise [50]. If there are more than one depth maps (multi-view), 3D points could be fused to increase the density, and decrease the error [6], [7], [18]. When there is only one view available, post-processing and regularization is used more frequently and visually coherent geometry is achieved [24], [26].
This particular workflow is giving us an important hint for the design, which is the fact that Depth prediction is not the final goal here, but an intermediate representation which allows a stable and scalable signal 2D to 3D geometry[24], [26], [50].

Camera intrinsics and pixel geometry. Let the camera intrinsics be represented by the matrix $\mathbf{K}$ (pinhole camera model) [4], [50]:

$$\mathbf{K} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}. \tag{2.1}$$

For an image pixel (u, v), define the homogeneous pixel vector [4], [50]:

$$\tilde{\mathbf{p}} = \begin{bmatrix} u \\ v \\ 1 \end{bmatrix}. \tag{2.2}$$

The corresponding normalized camera ray direction is obtained by [4], [50]:

$$\mathbf{r} = \mathbf{K}^{-1}\tilde{\mathbf{p}}. \tag{2.3}$$

Expanding gives the standard normalized coordinates [4], [50]:

$$x_n = \frac{u - c_x}{f_x}, \qquad y_n = \frac{v - c_y}{f_y}. \tag{2.4}$$

Thus the ray can be written as [4], [50]:

$$\mathbf{r} = \begin{bmatrix} x_n \\ y_n \\ 1 \end{bmatrix}. \tag{2.5}$$

Back-projection: depth map to 3D points. Assuming the depth map encodes the point's distance along the camera z-axis, the 3D point in camera coordinates is [4], [50]:

$$\mathbf{X}_c(u, v) = D(u, v) \cdot \mathbf{r} = D(u, v) \begin{bmatrix} x_n \\ y_n \\ 1 \end{bmatrix}. \tag{2.6}$$

Equivalently, the scalar form is [4], [50]:

$$X_c = \frac{(u - c_x)}{f_x} D(u, v), \quad Y_c = \frac{(v - c_y)}{f_y} D(u, v), \quad Z_c = D(u, v). \tag{2.7}$$

Validity masking. In practice, reconstruction is performed only over valid depth pixels [50]:

$$M(u,v) = \begin{cases} 1, & \text{if } D(u,v) > 0 \text{ and finite} \\ 0, & \text{otherwise.} \end{cases} \tag{2.8}$$

Resolution changes and intrinsics scaling. If the model predicts depth at a resized resolution (W', H') from an original image resolution (W, H), define:

$$s_x = \frac{W'}{W}, \qquad s_y = \frac{H'}{H}. \tag{2.9}$$

The intrinsics must be scaled consistently [50]:

$$f'_x = s_x f_x, \quad f'_y = s_y f_y, \quad c'_x = s_x c_x, \quad c'_y = s_y c_y. \tag{2.10}$$

and:

$$\mathbf{K}' = \begin{bmatrix} f'_x & 0 & c'_x \\ 0 & f'_y & c'_y \\ 0 & 0 & 1 \end{bmatrix}. \tag{2.11}$$

Multi-view point fusion (when multiple depth maps exist). For N views, the reconstructed world-space points from each view can be combined [6], [7], [18]:

$$\mathcal{P} = \bigcup_{i=1}^{N} \{\mathbf{X}_w^{(i)}(u,v) \mid M_i(u,v) = 1\}. \tag{2.12}$$

A common fusion strategy is voxel-based aggregation. With voxel size s, a point $\mathbf{X} = [x, y, z]^T$ maps to voxel index:

$$\mathbf{k} = \left\lfloor \frac{1}{s}\mathbf{X} \right\rfloor = \begin{bmatrix} \lfloor x/s \rfloor \\ \lfloor y/s \rfloor \\ \lfloor z/s \rfloor \end{bmatrix}. \tag{2.13}$$

The fused point for each voxel is the centroid [9]:

$$\bar{\mathbf{X}}(\mathbf{k}) = \frac{1}{|\mathcal{V}_\mathbf{k}|} \sum_{\mathbf{X} \in \mathcal{V}_\mathbf{k}} \mathbf{X}. \tag{2.14}$$

and the fused point cloud is [9]:

$$\mathcal{P}_{\text{fused}} = \{\bar{\mathbf{X}}(\mathbf{k}) \ \forall \mathbf{k}\}. \tag{2.15}$$

Mesh construction from depth (optional). If depth is dense on a grid, a surface can be created by triangulating neighboring pixels [50]. To avoid incorrect bridging across depth discontinuities, triangle formation is gated using a threshold τ [50]:

$$|D(u,v) - D(u+1,v)| > \tau \Rightarrow \text{do not connect that edge.} \tag{2.16}$$

2.1.2 Physics and Geometry Foundations

Depth-centric reconstruction is built on the geometric relationship between image coordinates and 3D coordinates [4], [50]. A 2D pixel corresponds to a ray in 3D space, and the depth value selects a specific point along that ray [4], [50]. This mapping requires camera parameters (intrinsics such as focal length and principal point) and a consistent coordinate convention [4], [50].

Rigid pose transformation (camera ↔ world). Let the camera pose in world coordinates be represented by a rotation matrix $\mathbf{R} \in SO(3)$ and a translation vector $\mathbf{t} \in R^3$. The homogeneous transform is [4], [50]:

$$\mathbf{T}_{wc} = \begin{bmatrix} \mathbf{R} & \mathbf{t} \\ \mathbf{0}^T & 1 \end{bmatrix}. \tag{2.17}$$

A camera-space point $\tilde{\mathbf{X}}_c = [X_c, Y_c, Z_c, 1]^T$ is mapped to world coordinates by [4], [50]:

$$\tilde{\mathbf{X}}_w = \mathbf{T}_{wc}\tilde{\mathbf{X}}_c. \tag{2.18}$$

Equivalently (non-homogeneous) [4], [50]:

$$\mathbf{X}_w = \mathbf{R}\mathbf{X}_c + \mathbf{t}. \tag{2.19}$$

The inverse transform (world to camera) is [4], [50]:

$$\mathbf{T}_{cw} = \mathbf{T}_{wc}^{-1} = \begin{bmatrix} \mathbf{R}^T & -\mathbf{R}^T\mathbf{t} \\ \mathbf{0}^T & 1 \end{bmatrix}. \tag{2.20}$$

so:

$$\mathbf{X}_c = \mathbf{R}^T(\mathbf{X}_w - \mathbf{t}). \tag{2.21}$$

Forward projection (geometry consistency). Given a 3D point in camera coordinates $\mathbf{X}_c = [X_c, Y_c, Z_c]^T$, the pixel projection satisfies [4], [50]:

$$\tilde{\mathbf{p}} \sim \mathbf{K}\mathbf{X}_c. \tag{2.22}$$

Expanded [4], [50]:

$$u = f_x\frac{X_c}{Z_c} + c_x, \qquad v = f_y\frac{Y_c}{Z_c} + c_y. \tag{2.23}$$

In multi-view reconstruction, viewpoint diversity provides parallax that stabilizes triangulation [4], [23], while single-view reconstruction must rely on learned priors or assumptions [24], [26]. Depth errors propagate through the camera model and produce visible artifacts after back-projection and meshing [50].

2.1.3 Ambiguity in 2D-to-3D

A core difficulty in 2D-to-3D reconstruction is ambiguity: multiple 3D scenes can produce similar or even identical 2D projections [4], [50]. From a single image no unique scale, absolute depth and hidden surfaces can be recovered without extra information [50].Occlusions add geometry that isn't present in the original scene,

textures can suggest absence of geometry and lighting or shading effects can suggest the absence of geometry[50].
This ambiguity is the motivation for the use of priors and constraints. Classical approaches introduce priors by means of handcrafted assumptions, while learning-based ones this encoding of priors from training data [24], [26], [50]. In will give ambiguity: with depth centric pipelines, an ambiguity is often resolved by training models to predict plausible depth consistent with common structures of objects and scenes [21], [24], [26], [42]. However, despite the presence of learning ambiguities between such structures are visible above all at the boundaries of objects, thin structures, reflective. there are surfaces and areas with no textures [50].
From a system point of view, ambiguity does not only have an impact on the accuracy of prediction but also reconstruction stability. Ambiguous regions may be the cause of holes, floating surfaces, or inconsistent depth discontinuities and hence propagating into point cloud artefacts [7], [50]. Awareness of ambiguity as a structural challenge goes toward justifying both the use of learning based depth estimation and the need for the post processing or fusion Strategies in reconstruction stage [7], [24], [26].

2.2 Related Works

2.2.1 Geometry-Based Methods

Geometry-based reconstruction methods are based on multi-view constraints: by seeing a particular point from different perspectives, it is possible to triangulate its 3D location [4], [23]. The projection matrices are common when the triangulation formulation is used. For view i, the projection matrix is [4], [50].

$$\mathbf{P}_i = \mathbf{K}_i[\mathbf{R}_i \mid \mathbf{t}_i]. \tag{2.24}$$

Given a 3D point in homogeneous coordinates $\tilde{\mathbf{X}} = [X, Y, Z, 1]^T$, the projected pixel satisfies [4], [50]:

$$\tilde{\mathbf{p}}_i \sim \mathbf{P}_i\tilde{\mathbf{X}}. \tag{2.25}$$

Triangulation is a process that solves for the value of $\tilde{\mathbf{X}}$ best matching observed pixels between views, often minimizing reprojection error[4]. Modern SfM/ MVS systems put these ideas into practice[18], [22], [23].
The major strength of the geometry-based methods is interpretability and correctness under valid assumptions [4].However, these methods deteriorate when viewpoint diversity is constrained, when scenes do not contain texture and when dynamic and reflective surfaces violate correspondence assumptions[4], [50].

2.2.2 Photometric Cue-Based Methods

Photometric cue-based approaches infer depth using signals such as shading, texture gradients, defocus blur, and perspective cues [1], [50]. These methods rely on assumptions about lighting, material properties, or scene structure [50]. While they can be computationally efficient, real-world scenes often violate the underlying assumptions due to complex lighting, shadows, specularities, and diverse textures, leading to unstable depth estimates [50].

For depth-centric 2D-to-3D systems, photometric cue-based literature motivates the need for priors in single-view reconstruction, and highlights why purely heuristic priors are insufficient at scale [24], [26].

2.2.3 Depth-Image-Based Rendering (DIBR)

Depth-Image-Based Rendering (DIBR) is a pipeline where a depth map is used to reproject or warp pixels to synthesize new viewpoints [26], [41]. In the broader 2D-to-3D context, DIBR acts as a bridge between depth estimation and view synthesis: once depth is known, pixels can be lifted into 3D and projected into a novel camera [4], [50].
Given a pixel (u, v) and depth $D(u, v)$, compute $\mathbf{X}_c(u, v)$ by back-projection (Equations 2.6–2.7). Then transform the point into a new camera frame using a relative pose $(\mathbf{R}_{rel}, \mathbf{t}_{rel})$ [4], [50]:

$$\mathbf{X}'_c = \mathbf{R}_{rel}\mathbf{X}_c + \mathbf{t}_{rel}. \tag{2.26}$$

Finally project into the new view [4], [50]:

$$u' = f'_x \frac{X'_c}{Z'_c} + c'_x, \qquad v' = f'_y \frac{Y'_c}{Z'_c} + c'_y. \tag{2.27}$$

A key advantage of DIBR is that it directly operationalizes depth into a geometric transformation [26]. However, DIBR pipelines commonly suffer from artifacts such as disocclusion holes, depth discontinuity stretching, and boundary tearing [35], [41], [44]. These artifacts often require hole filling, smoothing, or inpainting strategies [35]. Errors in depth estimation amplify these issues, since incorrect depth shifts pixels to incorrect 3D positions [44].

2.2.4 Deep Learning Methods

UNet Model

The U-Net model is used for segmenting images into different regions, which is critical in medical imaging for the detection of regions of interest, such as tumors or other pathologies. The attention modules that are inside the U-Net Model focus on specific areas of the part of the image that usually contains significant information, which increases the accuracy of the model in differentiating between various tissues or conditions [17].

The U-Net model, as shown in the given diagram, represents a sophisticated convolutional neural network architecture where it is mainly applied to image segmentation tasks, with special efficacy as biomedical imaging applications. Characterized by its unique "U-shaped" shape, the U-Net has two main components: the path of contraction (encoder) and the expansion path (decoder) [40].

The contraction path is intended to encapsulate contextual information from the image, which is a series of convolutional and max pooling layers. Each convolutional layer generally follows a rectified linear unit (ReLU) and batch normalization (BN) which make non-linear transformation and normalization of the input easy data, respectively. The addition of max pooling layers is used to counteract the reduction

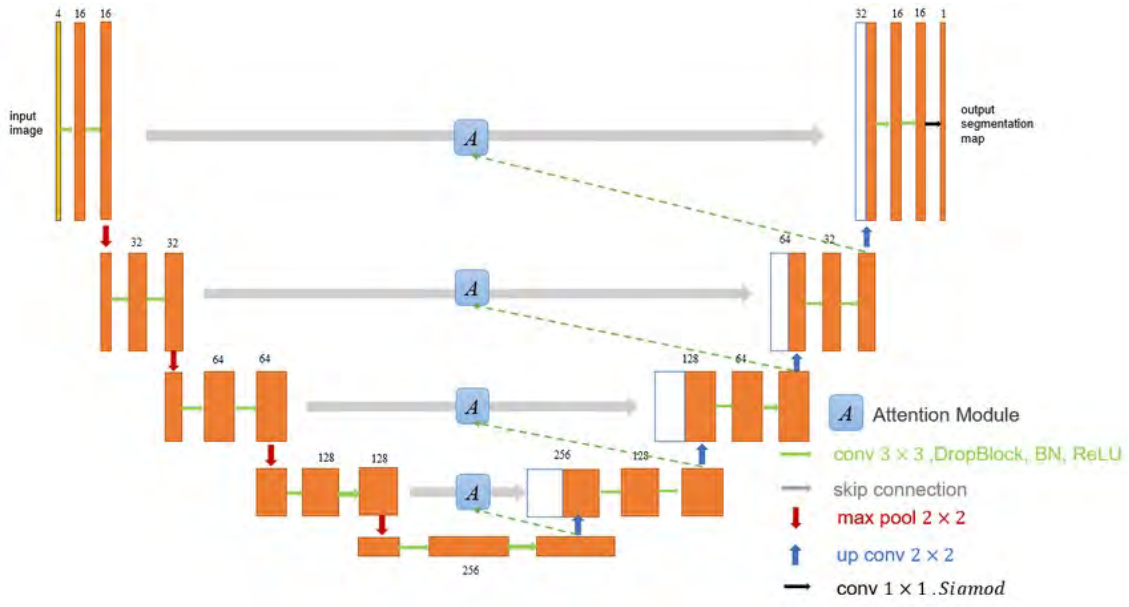


Figure 2.1: UNET Model Architecture [40]

in the spatial increases dimensions incrementally, thereby increasing the receptive field of the network while placing importance on key information [16].

As the opposite, the expansion path concentrates on achieving the localization required for meticulous segmentation. This pathway uses convolutions that are transposed to build up the feature maps to higher resolutions. An important part of this path is the inclusion of skip connections, which concatenate features between the contraction path and the expansion path as they exit the contraction path and input corresponding layers in the expansion path. These connections contribute to a fusion of global context information and local upsampled features, which further improves the ability of the network for accurate local predictions[16].

At the end stage of the network 1x1 convolution is used to map each high dimensional feature vector to that many number of classes. This produces a segmentation map that has a close correspondence to the spatial dimensions of the initial input image, which allows for pixel-wise exact classification [16]. The U-Net architecture also uses attention modules and DropBlock regularization [40]. Attention modules help the network to highlight salient features in an image in a strategic way, which potentially improves the accuracy of the partition. DropBlock acts as a structured dropout mechanism that selectively de-activates contiguous regions of the feature map and so forces the network to adapt itself by learning more robust features, which increases generalization [40].

ResNet18 Model

ResNet18 is a residual convolutional neural network architecture training to learn deep feature hierarchies but the stable gradient flow remains stable through identity based skip connections [51]. The key ability of ResNet like network is that it symbolically learn a ”residual mapping” instead of mapping directly, so that it can ease the

optimization difficulty and get deeper feature extractions without serious degradation. Because of its relatively light-weight depth and computation profile, ResNet18 is popularly utilized as backbone encoder in dense prediction tasks, namely, monocular depth estimation [32], [36], [47].

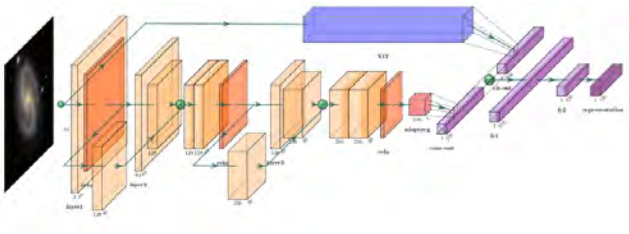


Figure 2.2: ResNet18 Model Architecture [51]

The ResNet18 architecture is structured as a sequence of convolutional stages that progressively downsample the spatial resolution while increasing channel capacity. It typically begins with a large-kernel convolution (followed by normalization and non-linearity) and a pooling layer, after which it enters four residual stages. Each stage is composed of stacked residual blocks, where the output of a small convolutional sub-network is added to an identity shortcut connection. This additive fusion preserves low-level information while allowing higher-level semantic features to emerge, making the encoder effective for extracting multi-scale representations needed for depth prediction [19].

In depth prediction pipelines, ResNet18 is commonly used as the encoder component of an encoder–decoder network, where the encoder compresses the input image into progressively abstract feature maps and the decoder upsamples these features back to a dense per-pixel depth map. A standard design is U-Net style decoding with skip connections, where intermediate feature maps from earlier ResNet stages are forwarded into the decoder to restore fine spatial details during reconstruction. This layout is particularly aligned with monocular depth estimation because depth is a dense output that requires both global context (scene structure) and local detail (edges and boundaries).

ResNet18 has been explicitly validated for monocular depth estimation in self-supervised settings. In Monodepth2, a ResNet18 encoder is paired with a U-Net style decoder and trained using photometric reprojection constraints rather than direct depth labels, enabling single-image depth inference at test time [36]. The authors emphasize that ResNet18 provides a strong balance between accuracy and efficiency, noting that it is substantially smaller than heavier backbones commonly used in earlier work [36]. Beyond self-supervision, ResNet-based backbones are also used in supervised monocular depth estimation models such as Deep Ordinal Regression Network (DORN), where the encoder extracts hierarchical features and depth is learned through an ordinal regression formulation that improves convergence and prediction stability [32].

From a computational perspective, ResNet18 is considered a lightweight backbone compared to deeper residual variants (e.g., ResNet50/101). In practical implementations used for dense prediction, it typically operates as an efficient feature extractor

with a relatively small parameter budget while still producing strong multi-scale representations. For example, Monodepth2 reports using a ResNet18 encoder with roughly 11M parameters in their depth network design [36]. Standard ResNet18 complexity is also commonly reported around 11.7M parameters and approximately 1.8G FLOPs (for ImageNet-scale inputs), which makes it feasible for sustained inference and training in resource-constrained environments [52]. This efficiency is one reason ResNet18 appears in improved depth architectures such as HR-Depth, where ResNet18 serves as the encoder while enhanced feature fusion and skip designs improve high-resolution depth boundaries [47].

GAN Model

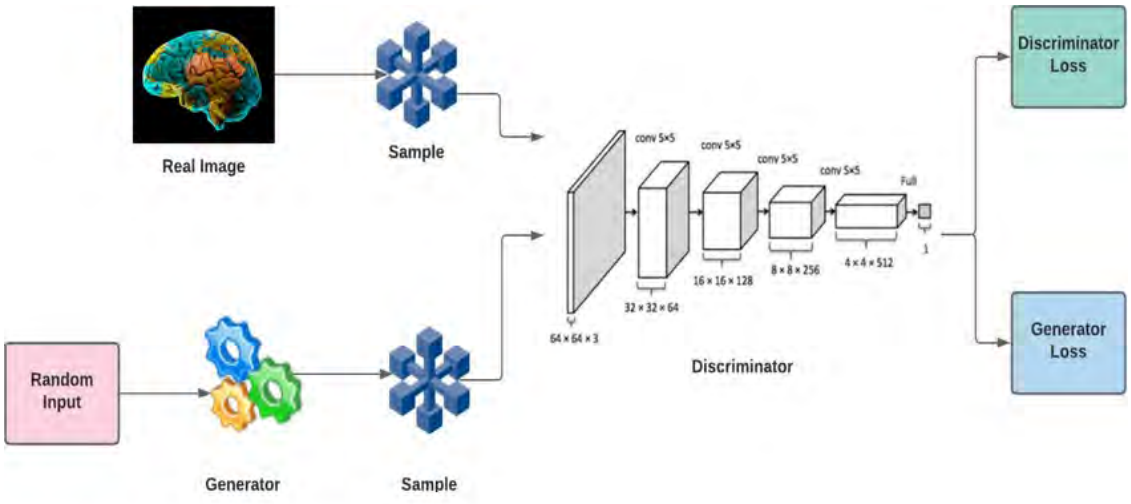


Figure 2.3: Block diagram of the Generative Adversarial Network (GAN) [46]

The Generative Adversarial Network (GAN) operates through a dynamic interaction between two neural networks: the generator and the discriminator. This model is particularly effective for tasks like converting 2D images into 3D models by learning to distinguish between real and synthetic data points [13].

The generator [G] starts the process by taking as input a random noise vector. This seed is used to produce an output similar to real data - in this case we are building a 3D model given a 2D image. The main aim of the generator is to deceive the discriminator by generating data which look as realistic as possible [13].

On the other hand, the discriminator [D] considers if a given input, which could be an real 3D model from the dataset or the synthetic model generated by the generator, is real or generated. It outputs the probability that the input is genuine, with the goal of properly classifying actual and synthetic data. The better the discriminator gets at identify satisfactory or genuine data, the more it forces the generator to enhance its outputs [13].

Both networks are taught at the same time in what is known as adversarial training. This training takes place in the context of a zero-sum game, where the generator is

trying to maximize the discriminator's probability of making a wrong classification, and former tries to keep its classification errors as low as possible. They use the gradients from their respective loss functions to update their weights - binary cross-entropy loss is usually used for the discriminator, and a loss measure that measures the generator's success in fooling the discriminator [12].

Throughout training, both the generator and the discriminator are improved. The generator generates data that are becoming more realistic, and the discriminator becomes better able to detect counterfeit data from real data. This iterative improvement runs until some stopping condition is reached such as reaching a set number of epochs or the stabilization of loss [15].

In the case of converting 2D images to 3D models by using GANs, the generator would be required to interpret 2D images (or their feature representations) and trying to create corresponding 3D models. These would then be evaluated by the discriminator models against real 3D data, which helps it get better at differentiating authentic models over time. Architectures such as UNet could be used within the generator for better handling of spatial hierarchies, which is very important to understand the transformation from 2D to 3D, all trained using adversarial methods, in an iterative fashion. This technique is contingent on having a good dataset that matches 2D images to 3D models or has an effective discriminator that is being trained to be able to recognize the features of 3D models associated with 2D images.

GNN Model

The Graph Neural Network (GNN) works by processing graph structured data through an iterative message-passing mechanism by which nodes are able to aggregate information from their neighbours. This model is especially good at doing things like node classification, link prediction and graph-based reasoning, where the learning from relational data is crucial [45]. GNNs have broad potential applications that span across the fields of such as social networks, molecular modelling, recommender systems, and knowledge graphs [49].

The process in a GNN layer is initialized by taking a set of node characteristics along with an adjacency matrix of a graph as an input. These features are iteratively up regulated as nodes exchange their information with their neighbours after the graph convolutional operations [20]. The principle goal on this process is generate node representations capture both the local as well as the global dependency in structure of this graph This is akin to how convolutional neural networks (CNNs) works with image except GNNs works of not euclidean (irregular) datatypes [45]. Having said that, on the other hand in Graph Attention Networks (GATs) which is an advanced version of GNNs using an attention mechanism to weigh the importance of the neighbouring nodes in the process of aggregating the information. Instead of treating all neighbour the same way GATs learns about adaptive weights, that allows to build most relevant relationships in the network. It is because of this self attentional approach performs highly in tasks in which the importance of the nodes shows to be varying on the graph [49].

GNNs are trained in an iterative optimization process which usually rely on semi-supervised Learning - relatively few nodes are created as being labeled [20]. The

Figure 2.4: Graph neural network (GNN) model structure[53]

model optimizes some sort of loss function : This is usually some sort of classification loss such as log cross entropy: It is a measure of the difference between the model predictions and the actual label. Back propagation and stochastic in gradient descent (SGD) or some variant of it is used to update the parameters of the network of such a number of epochs updating representations of the nodes at every step [45].

Throughout training GNNs learn to improve embeddings of nodes taking structural and feature based information; As the training process goes on, the model learns quality nodes representations in order to do more effective in the tasks of classification and prediction [20]. This iterative refinement process is continued till some convergence criteria is met, which may be stabilization of loss or number of training epochs [45].

In case of tasks associated to the vision (with the help GNNs), such as (recognition and scenes) graph generation, GNNs can be possibly used for processing images by representing images as graphs. For example, In Vision GNN (ViG) based architectures, images are converted into a graph in which the patches of pixels are considered to be being the nodes, and the edges are relations between them [49]. The model then performs some convolutions on the graphs, or attention-based message passing in order to extract high level semantic information that lead to better recognition accuracy. Similar to the way GAN's are utilising UNet Architectures to perform Image to Image tasks, ViG based models contain GNN based layers for the capturing spatial hierarchies effectively [49].

This technique is based on structurally well-grounded datasets, and where graphs are well-defined, and informative node features are present. Also in case of large-scale graphs GNNs require the use of good neighborhood sampling methods so as to efficiently process. Sapience Information, no computation cost [45]. In real-world scenarios, GNNs are used on a regular basis with regards to recommender systems (Eg, in ecommerce or social media platforms), where they learn from the graphs such represent the interactions of users and items, in order to improve the recommendations provided based on the relationships (infered) between users and products [53].

Overall, GNNs are a game-changing breakthrough for machine learning on graph data and provide a scalable and efficient method of learning on graph data. The graph operation of convolution operation & semi-supervised learning techniques allows the model that can make use of the local neighborhood information and global

graph structure, which gets very successful node representations [20]. The localized approximation to the spectral convolutions make GNNs computationally feasible for large graphs, in spite of being flexible, so that they can be used in wide range of domains. From social network analysis and modeling biological networks, GNN are where it is proving to be powerful tool in solving real world problem which involves complex, non-Euclidean data. As we learn more, the model will likely change to include even more complex tasks, paving the way for more advanced graph based techniques pertained to Machine Learning that will allow to tackle the challenges constituted by dynamic, heterogeneous and high dimensional graph data [45].

Comparison

The comparison of different 2D to 3D conversion technologies (U-Net, VAE, GAN, GNN and ResNet18) has shown their unique characteristics in terms of complexity, costs, their learning capacities and efficiency. U-Net is known as simple and efficient and can be used for real-time applications with limited computational resources. VAE although being complex and expensive, works best when producing new data from learned distributions, or applications where it is important to produce high fidelity generations. GANs have outputs which are highly realistic due to their excellent learning capacity but not efficient due to its intensiveness in training needs. GNN's provides adaptability to graph structured data with variable efficiency, MSDNNs works with multi scale data well however at the expense of large computation cost. The choice of which model to use is determined by some application requirements in a trade-off between the computational resources and accuracy.

Table 2.1: Comparison of 2D to 3D Conversion Technologies

Model	**Complexity**	**Computational Cost**	**Implementation Cost**	**Learning Capacity**	**Efficiency**
U-Net	Low	Low	Low	Good	High
VAE	Moderate-High	Moderate-High	Moderate	Excellent	Moderate
GAN	High	High	High	Excellent	Low
GNN	Moderate-High	Variable	Moderate-High	Good	Variable
ResNet18	Moderate	Low-Moderate	Moderate	Good	High

2.3 System Architectures

This section reviews the system-level design patterns that enable depth-centric 2D-to-3D reconstruction to run reliably at scale. While reconstruction accuracy is driven by the underlying algorithms and learning models, the ability to sustain throughput over long experiments depends heavily on how data are staged, moved, cached, and processed under constrained hardware budgets [34], [43]. In practice, depth prediction and reconstruction pipelines are dominated not only by GPU computation, but also by I/O latency, CPU preprocessing, memory pressure, and synchronization

overheads that can silently reduce utilization and destabilize training or evaluation [48]. As a result, architecture is treated in the literature as a first-class contributor to system performance rather than a secondary implementation detail [38].

Depth-centric pipelines are particularly sensitive to pipeline design because they combine multiple heavy stages: (i) data generation or dataset preparation, (ii) preprocessing and tensor formatting, (iii) GPU training or inference for dense depth prediction, and (iv) post-processing and geometric lifting into 3D representations [34], [43]. Without deliberate overlap between these stages, the overall runtime becomes limited by the slowest stage and the GPU can remain underutilized despite high theoretical compute capacity [38]. Consequently, modern systems frequently adopt staged execution, asynchronous loading, and cache-aware workflows to keep compute units saturated and to prevent repeated recomputation across epochs or checkpoints [38], [48].

The remainder of this section focuses on three recurring architecture choices that consistently appear in scalable reconstruction systems: SSD-backed caching for intermediate artifacts, producer–consumer style pipelines (often extended with explicit cleanup stages to control storage growth), and GPU acceleration via CUDA for reconstruction primitives that are inherently parallel [33], [38], [39]. These design patterns form the engineering backbone that makes depth-centric reconstruction feasible under realistic resource constraints.

2.3.1 System Design Overview

Reconstruction systems that operate at dataset scale are best understood as pipelines rather than monolithic programs. The literature commonly decomposes such systems into separable stages—data access, preprocessing, model execution, and reconstruction output—because each stage has distinct compute and memory characteristics and benefits from different optimization strategies [38]. For example, data loading and decoding are often CPU- and I/O-bound, while CNN-based depth inference is GPU- and memory-bandwidth-bound, and geometric lifting may shift between GPU and CPU depending on implementation [34], [43].

A key system objective is to overlap these stages so that slow operations (e.g., disk reads, image decoding, tensor packing) do not stall GPU execution [38]. In throughput-oriented training and evaluation, architectures therefore prioritize asynchronous execution, buffering, and clear separation of responsibilities between components. This separation improves utilization, reduces end-to-end latency variance, and increases reproducibility by making each stage easier to profile and control [43], [48].

In depth-centric reconstruction specifically, the design is often motivated by the cost structure of the pipeline. Depth inference is expensive but predictable, while upstream preparation and downstream reconstruction can become bottlenecks if repeatedly recomputed or if their data formats create excessive CPU–GPU synchronization [34], [48]. For this reason, the literature emphasizes architecture patterns that reduce repeated work and stabilize throughput, most notably caching and pipeline decoupling [38], [39].

2.3.2 Cache System in SSD

Caching is a standard systems strategy for reducing repeated computation and isolating expensive upstream stages from downstream experimentation [39]. In depth-centric 2D-to-3D pipelines, caching commonly stores preprocessed tensors, camera metadata, predicted depth maps, validity masks, and intermediate geometry (e.g., partial point clouds) so that training, ablation studies, and checkpoint evaluation can reuse identical inputs without rerunning rendering or preprocessing [38], [48]. This is particularly valuable when dataset preparation is costly (e.g., synthetic rendering, heavy augmentation, or multi-stage formatting), because recomputation introduces both runtime overhead and throughput variance across runs [38], [43].

SSD-backed caches represent a practical compromise between speed and capacity. Compared to HDD storage, SSDs provide significantly higher random-access performance and throughput, while remaining far more scalable than RAM in capacity for large datasets and long experiments [39]. This makes SSD caches well suited for storing mid-sized intermediate artifacts that must persist across multiple training epochs or across repeated evaluations over many checkpoints [48]. In addition, SSD caching supports reproducibility: when cached artifacts are treated as immutable once written, downstream experiments become deterministic with respect to data staging, enabling fair comparisons across model variants [38], [43].

Robust cache design also requires explicit lifecycle management and integrity signaling. The literature commonly uses state markers (folder conventions, manifest files, or completion flags) to distinguish between partially produced and fully valid cached outputs, preventing consumers from reading corrupted or incomplete artifacts [38]. This becomes important in long-running pipelines where interruptions (crashes, timeouts, out-of-disk events) are unavoidable. Clear cache states reduce failure propagation and support safe restart behavior, which is essential for scalable reconstruction experiments [38], [48].

2.3.3 Producer–Consumer–Destroyer in Deep Learning Pipelines

Producer–consumer architectures are widely used in high-throughput machine learning systems to decouple data preparation from model computation [38]. In this design, producers are responsible for generating or loading samples, performing preprocessing, and writing them into a buffer (often an on-disk cache in reconstruction workflows). Consumers then read prepared samples and execute GPU-heavy training or inference. This decoupling reduces GPU idle time caused by I/O stalls and allows each stage to be optimized independently [38], [48].

Depth-centric pipelines benefit strongly from this structure because CNN depth inference has predictable compute cost, while preprocessing and dataset staging often include variable-latency operations such as image decoding, resizing, normalization, mask preparation, and metadata parsing [43]. If these are performed inline on the same thread as GPU execution, transient slowdowns can reduce GPU utilization and increase wall-clock time significantly. Producer–consumer separation addresses this by allowing producers to run ahead, preparing data asynchronously so that the consumer can proceed at full speed [38].

The "destroyer" stage extends the classic producer–consumer pattern to enforce storage sustainability in long-running experiments [38]. In batch- or shard-based pipelines, cached data can expand quickly and exceed available disk space, especially when intermediate artifacts include multiple modalities (RGB, depth, masks, metadata) or repeated outputs across checkpoints. The destroyer deletes intermediate shards once they are consumed, maintaining bounded disk usage without requiring large RAM buffers or oversized storage [38], [48]. This is particularly relevant for scalable reconstruction workflows where experiments may run for many hours or days and must remain robust under strict storage limits [43], [48].

2.4 CUDA Programming

CUDA (Compute Unified Device Architecture) is NVIDIA's parallel programming model for executing general-purpose computation on GPUs. In reconstruction and deep learning pipelines, CUDA serves as the low-level execution substrate that enables dense, data-parallel workloads (image tensors, feature maps, depth buffers) to run efficiently on GPU hardware [30]. While high-level frameworks hide most kernel-level details, the computational behavior of CNN-based depth estimation and depth-to-geometry lifting is ultimately determined by CUDA's thread hierarchy, memory model, and synchronization rules.

Execution model (kernels, grids, blocks). CUDA programs offload work to the GPU by launching *kernels*, which execute the same function across many lightweight threads concurrently. Threads are organized into *blocks*, and blocks form a *grid*. This hierarchy maps naturally to vision workloads: a thread can process a pixel, a feature activation, or a depth value, enabling massive parallelism in per-element operations common in depth-centric pipelines [30]. Convolution-heavy CNN inference further aligns with this model because the same operations are applied repeatedly over regular 2D grids.

Memory hierarchy and performance implications. CUDA exposes multiple memory spaces with different latency and capacity trade-offs: global memory (large but high-latency), shared memory (small but fast, shared within a block), registers (fastest, per-thread), and read-only cached spaces such as constant and texture memory [34]. Efficient GPU utilization depends on how well memory access patterns match hardware constraints. CNN depth estimation benefits structurally because its tensors are dense and contiguous, enabling coalesced memory access and high reuse within optimized library kernels. In contrast, irregular 3D structures (graphs, meshes, adaptive sampling) tend to produce uncoalesced accesses and divergent control flow, reducing throughput and increasing overhead [31].

CUDA libraries as the practical foundation of deep learning. Most deployable depth estimation systems do not implement convolutions or tensor operations as custom kernels. Instead, they rely on vendor-optimized CUDA libraries[10]. cuDNN provides high-performance implementations of convolution, pooling, normalization, and activations; cuBLAS accelerates dense linear algebra operations[3], [10]. Encoder–decoder CNNs map almost entirely onto these primitives, which is a key reason depth-centric CNN pipelines achieve low overhead without bespoke CUDA development[10], [37]. This library alignment is one of the main system-

level advantages of CNN-based depth estimation compared to heavier 3D-first methods[27], [28].

Framework integration (PyTorch / LibTorch). Modern deep learning systems typically access CUDA through frameworks such as PyTorch and its C++ frontend (LibTorch). These frameworks dispatch operations to optimized CUDA kernels, manage GPU memory allocation, and support mixed-precision execution with minimal code changes. For depth-centric reconstruction, this abstraction layer improves reproducibility and maintainability: training, evaluation, and deployment can share the same tensor semantics and checkpoint formats, while still benefiting from CUDA-optimized execution.

Mixed precision and tensor-core acceleration. Modern GPUs include specialized hardware for reduced-precision matrix operations. Mixed-precision training and inference use FP16/BF16 for most computation while preserving higher precision where numerically necessary, increasing throughput and reducing memory footprint. Depth estimation CNNs are especially compatible with mixed precision because they are convolution-heavy and operate within bounded numeric ranges when properly parameterized, enabling higher resolution or larger batch sizes under fixed GPU memory budgets [34].

Streams, overlap, and pipeline throughput. In system-scale reconstruction workflows, overhead is often dominated by data movement and synchronization rather than raw FLOPs. CUDA streams enable overlapping host-to-device transfers, preprocessing, and inference so the GPU remains saturated. Depth-centric pipelines are well suited to this execution style because preprocessing (resize/normalize), inference, masking, and even depth-to-point unprojection can be implemented as GPU-resident operations, minimizing CPU–GPU round trips and avoiding frequent synchronization barriers.

Profiling and practical bottleneck diagnosis. Achieving low overhead requires measurement-driven optimization. CUDA profiling tools (e.g., Nsight Systems and Nsight Compute) expose kernel timelines, memory transfer patterns, and utilization behavior. In many depth-centric pipelines, profiling reveals bottlenecks such as redundant memory transfers, excessive kernel launch overhead from fragmented operators, or unnecessary CPU synchronization during post-processing. Because CNN-based depth estimation has a predictable computation graph, these bottlenecks are easier to isolate and fix than in pipelines with iterative sampling or irregular control flow.

Relevance to 2D-to-3D reconstruction. CUDA directly supports two critical stages in depth-first 2D-to-3D systems: (1) *Depth inference*: CNN encoder–decoder inference is dominated by CUDA-accelerated convolution and tensor operations, benefiting from cuDNN and mixed precision [34]. (2) *Geometric lifting*: depth maps can be converted to 3D point clouds via per-pixel back-projection using known intrinsics. This is a regular per-element computation that maps cleanly to CUDA kernels, allowing depth-to-geometry conversion to remain on-device and avoid CPU bottlenecks. Keeping both stages GPU-resident reduces synchronization overhead and preserves end-to-end throughput in reconstruction pipelines.

CUDA matters in this thesis not as an isolated programming topic, but because it explains why depth-centric CNN reconstruction is system-efficient in practice. The

combination of a regular 2D tensor representation (depth as 2.5D), convolution-dominated inference, mature CUDA libraries, mixed-precision acceleration, and stream-based pipelining provides a stable, low-overhead execution path from RGB images to usable 3D geometry on commodity GPUs [31], [34].

Chapter 3

Methodology

This chapter describes the complete pipeline used in this thesis, from raw 3D assets to trained depth prediction and final 3D reconstruction. The methodology is designed to be (i) reproducible, through explicit dataset splits and manifests, and (ii) scalable, through a disk-backed producer consumer cache architecture that allows rendering and training to run continuously under limited GPU memory and storage constraints.

3.1 Dataset Preparation

3.1.1 Dataset Description

The core 3D dataset used in this work is ShapeNetCore, a large annotated repository of CAD-style 3D models organized by semantic WordNet synsets [14]. ShapeNetCore contains over 51,000 models across 55 object categories, where each model includes an aligned mesh and associated material definitions and texture images. In the original layout, each object category is stored under a synset directory (e.g., `02691156`), and each instance is stored under a unique model ID directory, typically containing a `models/` folder (mesh and material files) and an `images/` folder (texture images). A typical directory structure is:

```
<synset>/<model_id>/
  models/
    model_normalized.obj
    model_normalized.mtl
  images/
    texture0.jpg
```

This dataset is well-suited for depth-centric 2D-to-3D reconstruction because it provides clean 3D geometry that can be rendered from controlled camera configurations to generate aligned RGB–depth supervision. Using synthetic depth avoids sensor noise and permits consistent ground truth under known intrinsics and extrinsics, which is critical for later geometric back-projection and reconstruction.

3.1.2 Prepared Dataset Structure

Raw ShapeNetCore data is transformed into a prepared dataset layout to support efficient indexing, reproducible splits, and robust rendering. Performs the following

steps:

1. **Integrity validation:** Each model directory is checked for the presence of a valid mesh (`model_normalized.obj`) and its associated material file (`.mtl`). Incomplete or corrupted models are discarded to prevent renderer failures and training-time I/O errors.

2. **Split materialization:** Instead of relying on external lists, explicit `train/`, `val/`, and `test/` root folders are created. Each split mirrors the synset hierarchy and contains mutually exclusive model IDs, enabling deterministic evaluation.

3. **Texture path stabilization:** Texture images referenced by `.mtl` files are made directly resolvable from the model directory. Symbolic links are used to avoid duplication while preserving correct relative paths.

4. **Auxiliary file policy:** Only geometry, materials, textures, and optionally voxel files are retained. Extraneous screenshots or metadata not required for rendering or training are ignored to reduce storage overhead.

The prepared dataset uses the following split layout:

```
train/
  <synset>/
    <model_id>/
      models/model_normalized.obj
      models/model_normalized.mtl
      texture*.jpg   (symlink(s))
val/
  <synset>/
    <model_id>/ ...
test/
  <synset>/
    <model_id>/ ...
```

If volumetric occupancy files (`.binvox`) are available, they are preserved unchanged for completeness and potential future use in volumetric supervision or evaluation. Table 3.1 summarizes the transformation.

3.1.3 Training Split

The dataset is divided into three disjoint partitions: 80% training, 10% validation, and 10% test. The split is performed at the model instance level (model IDs), not at the rendered-view level, to prevent leakage where different views of the same underlying geometry appear across splits. Where multiple categories are used, the split is constructed to preserve class balance across synsets (stratified allocation). The final split is persisted by the folder structure itself, which guarantees reproducibility without relying on external CSV indices.

Table 3.1: Comparison of raw ShapeNetCore and prepared dataset structures

Aspect	**Raw (ShapeNetCore)**	**Prepared Dataset**
Categories	Synset folders with no explicit splits.	Same synsets partitioned under `train/`, `val/`, `test/`.
Mesh layout	`<synset>/<model_id>/models/` with `.obj/.mtl`.	Same mesh layout, but grouped by split for reproducible loaders.
Textures	Stored in `images/` and referenced by `.mtl`.	Texture symlinks placed where `.mtl` resolution is stable.
Splits	Not materialized unless external lists are used.	Explicit split folders; each model belongs to exactly one split.
Voxel data	Optional `.binvox` representations.	Retained unchanged for future geometry tasks.

Symlink Policy

Symbolic links are used to avoid duplicating large texture assets and to keep the prepared dataset compact. This is particularly important when train/val/test are materialized as separate directory trees; symlinking ensures that the `.mtl` file references resolve correctly while preventing unnecessary copies. This design also supports fast dataset scanning and simplified data loader logic, because each split can be enumerated as a directory tree.

3.1.4 Synthetic RGB–Depth Rendering

To train a depth prediction network, synthetic RGB–depth pairs are generated by an off-screen OpenGL renderer integrated into the pipeline. For each 3D model instance, multiple views are rendered from sampled camera poses. Each view yields: (i) an RGB image and (ii) an aligned depth map, stored together with the camera parameters required for geometric reconstruction.

Camera Model and Intrinsics

Each render uses a pinhole camera model with intrinsic matrix [4], [50]:

$$\mathbf{K} = \begin{pmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{pmatrix} \tag{3.1}$$

where f_x, f_y are focal lengths in pixels and (c_x, c_y) is the principal point. The intrinsics are stored per-view to ensure that downstream unprojection uses the exact parameters that produced the depth map.

Camera Rigs and View Sampling

To improve generalization and reduce viewpoint bias, the renderer supports multiple camera rig configurations. In this thesis, views are generated by sampling camera poses around the object, typically using orbit or hemisphere sampling, where camera azimuth and elevation are randomized within specified ranges and the camera radius

is chosen to frame the object consistently. Each pose is represented by a world-to-camera transform $\mathbf{T}_{wc} \in R^{4\times4}$ (or equivalently rotation $\mathbf{R}$ and translation $\mathbf{t}$), stored in the per-sample manifest.

Depth Calculation and Linearization

Depth maps are obtained from the OpenGL depth buffer (Z-buffer), which is non-linear with respect to camera-space distance due to perspective projection. Given near and far plane distances (n, f), the depth buffer value $z_b \in [0, 1]$ is converted to linear camera-space depth Z by applying the corresponding inverse projection mapping used by the renderer. The linear depth representation is used for learning and for later back-projection, because it has a direct metric interpretation along the camera forward axis. Background pixels (no surface hit) are assigned a fixed sentinel depth (e.g., 0) and/or a validity mask is stored to separate valid geometry from background.

Per-Sample Metadata (Manifest)

For each rendered sample, the pipeline writes a manifest entry containing:

- paths to RGB and depth files,
- resolution (W, H),
- intrinsics (f_x, f_y, c_x, c_y),
- camera extrinsics (pose), and
- depth conventions (near/far, background depth, and masking policy).

Storing these parameters alongside the images ensures strict consistency between rendering, training, inference, and reconstruction, and avoids mismatches caused by hard-coded camera settings in downstream tools.

3.2 Producer Consumer SSD Cache Architecture

Rendering large numbers of RGB–depth pairs can exceed RAM limits and can bottleneck GPU training if data is not available when needed. To sustain training throughput, the system uses a disk-backed producer–consumer architecture, where: producer renders new samples to an SSD cache, consumer trains on cached samples, and destroyer deletes samples that are no longer needed. Controller reads flags and controls the flow of command.

Making the SSD Act as Volatile Storage

The cache is treated as a *volatile* dataset store: only a bounded window of rendered samples is kept on disk at any time. Once a sample has been consumed by training and is no longer required, it is safely deleted to free SSD space. This enables continuous training even when the full rendered dataset would not fit on disk.

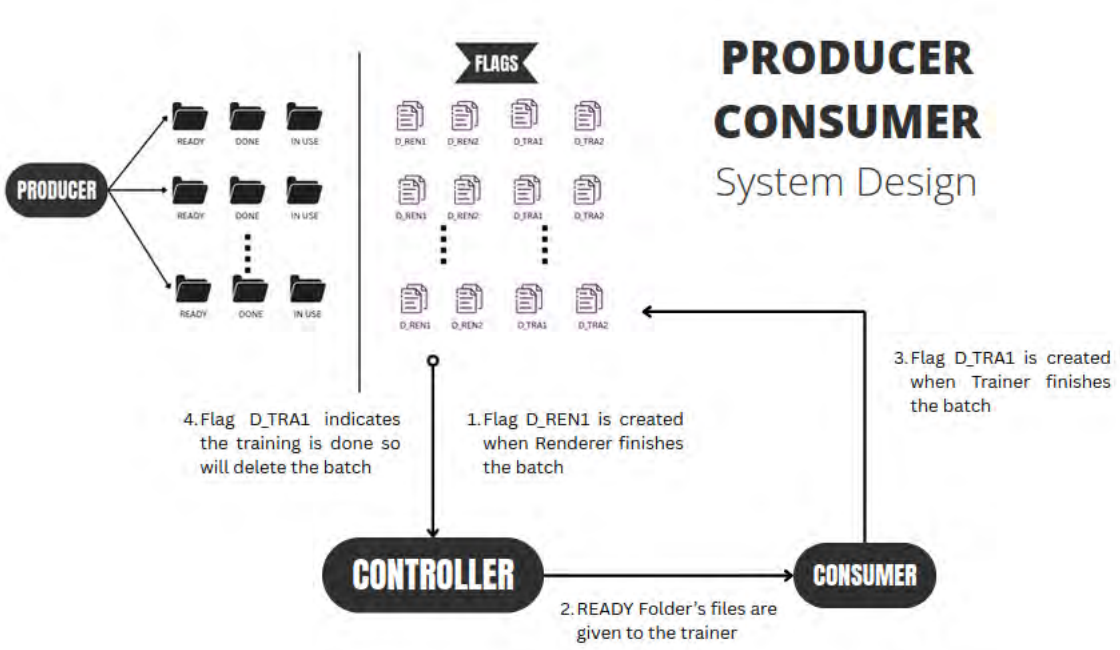


Figure 3.1: Producer Consumer influenced System Design

3.2.1 Cache States and Atomic Transitions

The cache is organized into explicit state folders, for example `READY`, `IN_USE`, and `DONE`. A sample moves across states via atomic filesystem operations (rename/move), which provides crash resilience and prevents partial consumption:

- **READY**: fully rendered samples awaiting training.
- **IN_USE**: samples currently being loaded/used for a training step.
- **DONE**: samples already consumed and eligible for deletion or archiving.

A manifest file (e.g., `manifest.json`) is written only after the sample is complete (RGB, depth, metadata), ensuring the consumer never trains on partially written data.

Table 3.2: SSD cache states and actions

State	Meaning	Action
READY	Sample complete and available.	Producer writes; consumer selects next batch from here.
IN_USE	Sample is locked for training.	Consumer moves here during batch load to avoid duplicates.
DONE	Sample consumed by training.	Destroyer deletes after safety conditions are met.

Done-Flag Mechanism for Batches and Epochs

Because the pipeline can generate and train on data in chunks, an explicit `DONE_RENDERING` barrier file is used to indicate that a chunk is complete and stable. This mechanism supports epoch-style training over chunked datasets:

1. The producer renders a bounded chunk of samples and writes `DONE_RENDERING_k` when finished.
2. The consumer trains over that chunk (one pass or multiple passes, depending on configuration).
3. After successful consumption, the destroyer deletes the chunk and the system advances to chunk k+1.

This design ensures that training never races ahead of rendering, avoids mixing incomplete chunks, and enables deterministic repetition when multiple epochs are required (by re-rendering or by retaining chunk manifests for repeated traversal, depending on SSD constraints).

3.3 Depth Prediction Model and Training

3.3.1 ResUNet18 Architecture

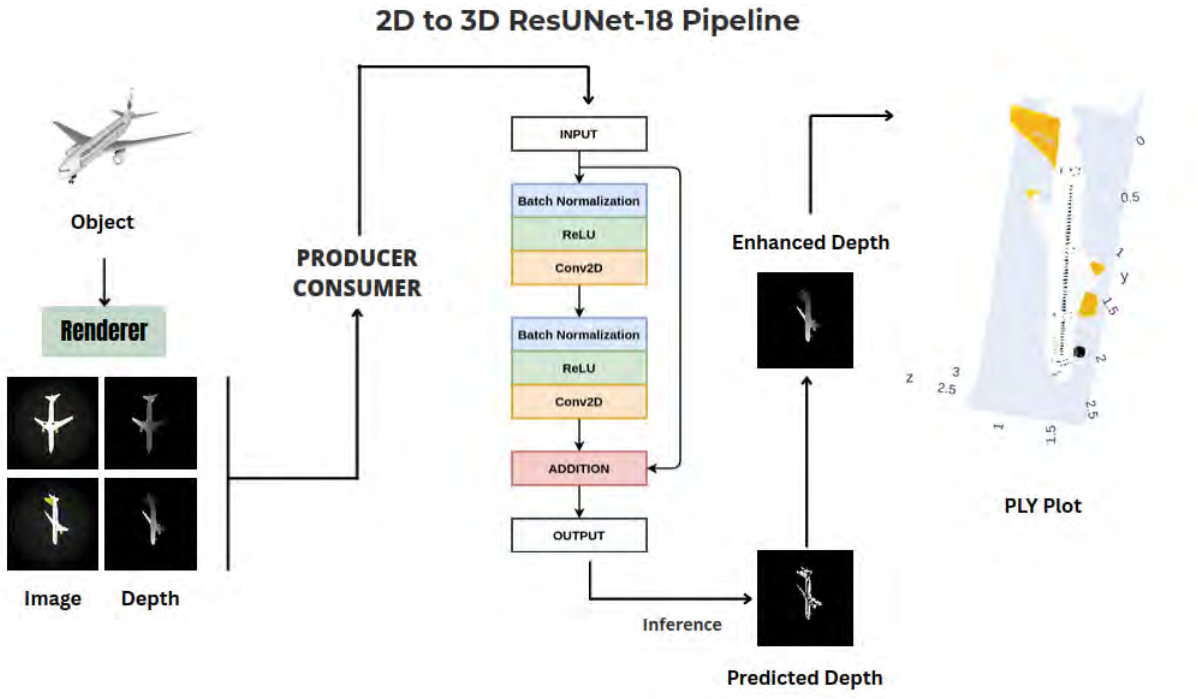


Figure 3.2: Pipeline Architecture

The depth prediction network is a ResUNet18: a U-Net style encoder–decoder where the encoder is a ResNet-18 backbone and the decoder performs progressive upsampling with skip connections. The encoder extracts hierarchical features at multiple scales, while skip connections fuse high-resolution spatial detail from early layers

into the decoder to preserve edges and fine structures. This design follows the residual U-Net principle of combining residual units (identity mappings) with an encoder–decoder topology and long skip connections between corresponding levels, which improves information propagation and supports accurate pixel-wise predictions [29]. The final layer outputs a single-channel dense depth map aligned with the input resolution.
In each residual unit, the output is expressed as an identity mapping plus a residual function, which eases optimization in deeper networks and mitigates degradation effects [29]. In the decoding path, feature maps are upsampled and concatenated with features from the corresponding encoding level, enabling the network to recover fine spatial details while retaining high-level context [29].
To enforce non-negative depth predictions, the network output uses a Softplus activation:

$$\text{Softplus}(x) = \ln(1 + e^x), \tag{3.2}$$

which yields strictly positive values while maintaining smooth gradients.

3.3.2 Training Objective

Let y_i denote the ground-truth depth at pixel i and $\hat{y}_i$ the predicted depth. Training minimizes a depth regression loss computed over valid pixels only (foreground or non-background), using a binary mask $m_i \in \{0, 1\}$. The number of valid pixels is $N = \sum_i m_i$. Masking invalid or missing depth values is important in practice because depth targets often contain gaps (e.g., near object boundaries and specular surfaces), so the loss is evaluated only on valid points [11].
A common choice for monocular depth is the scale-invariant logarithmic loss (SILog), which encourages correct *relative* depth structure:

$$\mathcal{L}_{\text{SILog}} = \frac{1}{N} \sum_i m_i (\log y_i - \log \hat{y}_i)^2 - \frac{1}{N^2} \left(\sum_i m_i (\log y_i - \log \hat{y}_i) \right)^2 . \tag{3.3}$$

This objective is consistent with the scale-invariant log-depth error introduced for monocular depth prediction, which emphasizes depth relations within a scene rather than global scale [11]. In addition, an L1 depth loss can be used to stabilize absolute error:

$$\mathcal{L}_{\text{L1}} = \frac{1}{N} \sum_i m_i \left| y_i - \hat{y}_i \right| . \tag{3.4}$$

The final objective is a weighted combination:

$$\mathcal{L} = \lambda_1 \mathcal{L}_{\text{SILog}} + \lambda_2 \mathcal{L}_{\text{L1}}, \tag{3.5}$$

where λ_1, λ_2 are set empirically.

Optimization and Data Loading

Training is implemented using a LibTorch-based pipeline. Mini-batches are loaded from the SSD cache, ensuring that GPU compute is continuously supplied with data while rendering proceeds in parallel. Standard optimization is performed using Adam with a fixed learning rate, and gradient clipping is applied when necessary to improve stability under mixed data distributions across chunks.

3.3.3 Evaluation Metrics

To evaluate depth prediction quality, metrics are computed on the validation and test splits using valid pixels only. The following metrics are typical and directly interpretable for reconstruction quality, and are standard in monocular depth evaluation [11]:

Root Mean Squared Error (RMSE).

$$\text{RMSE} = \sqrt{\frac{1}{N}\sum_i m_i(y_i - \hat{y}_i)^2}. \tag{3.6}$$

RMSE penalizes large depth errors, which often manifest as severe surface warping in 3D reconstruction.

Mean Absolute Relative Error (AbsRel).

$$\text{AbsRel} = \frac{1}{N}\sum_i m_i \frac{|y_i - \hat{y}_i|}{y_i}. \tag{3.7}$$

AbsRel captures relative consistency and is sensitive to scale distortions across the depth range.

Accuracy under threshold (δ).

$$\delta_i = \max\left(\frac{\hat{y}_i}{y_i}, \frac{y_i}{\hat{y}_i}\right), \quad \%(\delta < \tau) = \frac{1}{N}\sum_i m_i \, I\left[\delta_i < \tau\right], \tag{3.8}$$

where common thresholds are $\tau \in \{1.25, 1.25^2, 1.25^3\}$ [11]. These accuracy measures quantify how many pixels are within a bounded multiplicative error, which correlates with preserving shape proportions during reconstruction.

3.4 3D Reconstruction

The final stage lifts the predicted depth map into an explicit 3D geometric representation by applying a back-projection (inverse projection) per valid pixel. Back-projection is a standard operation in projective geometry: it converts an image measurement together with depth into a 3D point in the camera coordinate system [5]. This "depth-to-geometry" step is a core primitive used by dense RGB-D reconstruction systems and volumetric fusion methods, where depth maps are repeatedly converted into 3D samples and then integrated into a global model [2], [8].

In this work, the depth prediction network is trained to output *camera-centric* depth (i.e., depth values referenced to the camera coordinate frame). This enables direct geometric lifting from predicted depth to 3D without any additional learned coordinate transforms, and it aligns with the camera-centric depth representation used in classical reconstruction pipelines [5], [8].

Point Cloud Generation (Depth Unprojection)

Let $\hat{Z}(u, v)$ denote the predicted depth at pixel (u, v). For each valid pixel, the corresponding 3D point in the camera frame is obtained by back-projection:

$$\mathbf{P}_{\text{cam}}(u, v) = \hat{Z}(u, v) \cdot \mathbf{K}^{-1} \begin{bmatrix} u \\ v \\ 1 \end{bmatrix}, \tag{3.9}$$

where $\mathbf{K}$ is the camera calibration matrix associated with the rendered dataset. Expanding (3.9) yields:

$$X = \frac{(u - c_x)\hat{Z}}{f_x}, \qquad Y = \frac{(v - c_y)\hat{Z}}{f_y}, \qquad Z = \hat{Z}. \tag{3.10}$$

Equations (3.9)–(3.10) are the standard inverse of the pinhole projection mapping used throughout multi-view geometry and dense RGB-D reconstruction [8].

Validity Masking and Outlier Rejection

Not all pixels should be converted into 3D points. Background pixels, missing depth, and non-finite predictions are excluded to avoid outlier geometry. With a binary validity mask $m(u, v) \in \{0, 1\}$, points are generated only when $m(u, v) = 1$ and $\hat{Z}(u, v)$ is finite and within a plausible depth range. This filtering step matches standard practice in dense reconstruction systems, where invalid measurements are rejected prior to integration to avoid corrupting the resulting surface [2], [8].

3.4.1 Triangle Mesh Construction from Depth

While a point cloud is sufficient for visualization and evaluation, many applications require a connected surface representation. A common approach is to form a *triangle mesh* directly from the depth image by exploiting its regular 2D grid structure. Each pixel (u, v) corresponds to a 3D vertex $\mathbf{P}_{\text{cam}}(u, v)$, and each image-space quad

$$(u, v),\ (u + 1, v),\ (u, v + 1),\ (u + 1, v + 1)$$

is triangulated into two faces, provided that all participating vertices are valid.

Grid Connectivity and Face Emission

Define the four vertices of a quad:

$$\begin{aligned} \mathbf{p}_{00} &= \mathbf{P}_{\text{cam}}(u, v), & \mathbf{p}_{10} &= \mathbf{P}_{\text{cam}}(u + 1, v), \\ \mathbf{p}_{01} &= \mathbf{P}_{\text{cam}}(u, v + 1), & \mathbf{p}_{11} &= \mathbf{P}_{\text{cam}}(u + 1, v + 1). \end{aligned} \tag{3.11}$$

When all four vertices are valid, the quad can be triangulated as:

$$\Delta_1 = (\mathbf{p}_{00}, \mathbf{p}_{10}, \mathbf{p}_{01}), \qquad \Delta_2 = (\mathbf{p}_{10}, \mathbf{p}_{11}, \mathbf{p}_{01}). \tag{3.12}$$

This produces a watertight mesh over locally smooth regions of the depth map.

Depth Discontinuity Constraint

A naive grid triangulation will incorrectly connect foreground and background across occlusion boundaries, producing long "bridge" triangles. To prevent this, a depth discontinuity constraint is applied before emitting faces. Let

$$d(\mathbf{a}, \mathbf{b}) = \|\mathbf{a} - \mathbf{b}\|_2 \tag{3.13}$$

denote the 3D edge length between two neighboring vertices. A triangle is emitted only if all of its edges satisfy:

$$d(\mathbf{p}_i, \mathbf{p}_j) < \tau_{\text{edge}}, \tag{3.14}$$

where τ_{edge} is a user-selected threshold (e.g., proportional to the expected local sampling density). This rejects triangles that would span depth jumps and is consistent with the general principle in dense reconstruction of avoiding integration across discontinuities and outliers [2], [8].

Normal Estimation

Normals can be estimated per vertex using local finite differences on the depth grid. For example, using neighboring vertices:

$$\mathbf{n}(u, v) = \frac{(\mathbf{P}_{\text{cam}}(u+1, v) - \mathbf{P}_{\text{cam}}(u, v)) \times (\mathbf{P}_{\text{cam}}(u, v+1) - \mathbf{P}_{\text{cam}}(u, v))}{\|(\mathbf{P}_{\text{cam}}(u+1, v) - \mathbf{P}_{\text{cam}}(u, v)) \times (\mathbf{P}_{\text{cam}}(u, v+1) - \mathbf{P}_{\text{cam}}(u, v))\|_2}. \tag{3.15}$$

Normals improve rendering quality and can support downstream surface processing.

3.4.2 Computational Cost and CUDA Acceleration

Reconstruction from dense depth is computationally heavy because it requires *per-pixel* and *per-quad* processing. For an image of size $H \times W$, unprojection produces up to HW vertices, and triangulation considers approximately $(H-1)(W-1)$ quads, each requiring validity tests, discontinuity checks, and (potentially) emission of two faces. In multi-frame fusion settings, these steps must be repeated for many depth maps and are therefore a dominant computational cost in dense mapping systems [2], [8].

Despite the heavy workload, the pipeline is a sequence of deterministic mathematical operations with strong data parallelism:

- **Vertex generation:** each pixel maps independently to one 3D point via (3.9)–(3.10).
- **Face decisions:** each quad can be processed independently to decide whether to emit triangles via (3.12) and (3.14).
- **Normal estimation:** each valid vertex uses a small local neighborhood as in (3.15).

This independence makes the reconstruction stage well-suited for CUDA. A CUDA implementation assigns one GPU thread per pixel for unprojection and one GPU thread per quad for triangle eligibility tests and face generation. Such GPU acceleration is consistent with dense reconstruction systems that rely on parallel processing to achieve practical runtimes, including real-time dense surface tracking/mapping [8] and volumetric fusion approaches that integrate large numbers of depth samples efficiently [2].

Table 3.3: Hardware and software configuration used in this thesis

Component	**Value**
CPU	Intel Core i7-11800H (8 cores / 16 threads)
GPU	NVIDIA GeForce RTX 3050 (4 GB VRAM)
CUDA toolkit	CUDA 12.0
LibTorch (PyTorch C++ API)	**LibTorch 12.4**
Operating system	Ubuntu 22.04 LTS
Primary storage (dataset/cache)	SSD (disk-backed cache)

Memory and bandwidth considerations. The reconstruction workload is not only compute-heavy but also memory-bandwidth intensive: vertices, masks, and (optionally) normals and triangle indices must be written in contiguous buffers. CUDA helps by enabling coalesced memory access patterns when processing pixels in raster order, and by allowing prefix-sum (scan) based compaction to pack only valid vertices and faces into output arrays. This reduces wasted bandwidth on invalid/background pixels and supports scalable reconstruction at higher resolutions [2], [8].

3.5 Implementation

This section describes the engineering implementation of the proposed pipeline and explains how the system is designed to operate reliably on a medium-powered workstation. In many industrial 3D rendering and dataset generation workflows, large-scale rendering is performed on higher-end machines or compute clusters with substantial GPU memory, persistent high-throughput storage, and fully materialized datasets that remain stable across repeated experiments. In contrast, this thesis targets an environment with limited GPU memory and finite local storage, where rendering, training, and reconstruction must be executed as a controlled continuous process. The implementation demonstrates that, through chunk-based data streaming, deterministic re-rendering for epoch repetition, and GPU-parallel reconstruction, a complete 2D-to-3D pipeline can be executed reproducibly and at practical throughput on commodity hardware. The reconstruction stage uses CUDA programming, and experiments are conducted with CUDA 12.0.

3.5.1 Experimental Configuration and Parameters

This thesis is implemented and evaluated on a single workstation configuration. The overall pipeline parameters (dataset scale, rendering schedule, and training schedule) are reported together with the full set of training hyperparameters to ensure reproducibility. Table 3.3 summarizes the hardware/software environment, while Table 3.4 and Table 3.5 report the pipeline-level settings and the optimizer/loss configuration used during ResUNet18 training.

Table 3.4: Training and pipeline parameters used throughout dataset generation and learning

Stage	**Parameter**	**Value**
Dataset	Total 3D models	52,000
Dataset	Train / Val / Test split	80% / 10% / 10% (model-level split)
Rendering	Views per model	12
Rendering	Models per rendered chunk	100
Rendering	Samples per chunk	1,200 RGB–depth pairs
Rendering	Standby buffer	1 chunk ahead of consumer
Input	RGB/depth resolution	512×512

3.5.2 OpenGL-Based Synthetic RGB–Depth Rendering

Synthetic supervision is generated using an off-screen OpenGL renderer that produces aligned RGB images and depth maps. Rendering is performed at a fixed resolution of 512×512, ensuring consistent network inputs and a uniform grid for downstream reconstruction. Each CAD model is rendered from 12 viewpoints to increase viewpoint diversity and reduce bias, while maintaining a controlled camera-centric depth definition suitable for geometric lifting.

To scale rendering to approximately 52,000 models under limited storage, rendering is executed in bounded units. Each unit (chunk) contains 100 models, which yields 1,200 RGB–depth pairs per chunk. A standby chunk is maintained so that, while the learning stage consumes one chunk, the renderer can prepare the next chunk and reduce idle periods in the training stage.

3.5.3 Producer Consumer Execution

A key engineering contribution of the implementation is the use of a streaming producer–consumer workflow to support multi-epoch training without permanently storing the full rendered dataset. Rendering and training are treated as coordinated stages of a continuous process. Rendering produces a bounded chunk of fully specified samples (RGB, depth, and the associated metadata required for learning and reconstruction). Training consumes the chunk, and after successful consumption the chunk is deleted to reclaim storage. This design keeps the storage footprint approximately constant regardless of the total dataset size.

In conventional training, an epoch corresponds to one complete pass over a fixed dataset stored persistently on disk. In this thesis, because rendered data is treated as volatile to reduce overhead, epoch repetition is realized through deterministic re-rendering. Each epoch is defined as a full traversal of the training split, realized as a deterministic sequence of chunks (consistent model assignment and view sampling policy). When the system begins the next epoch, chunks are regenerated under the same rules. This preserves a stable epoch definition while enabling a 30-epoch schedule without requiring the entire dataset to be stored at once.

Correctness is maintained through explicit chunk completion signaling. Training

Table 3.5: Training hyperparameters and optimization settings

Hyperparameter	**Value**
Optimizer	Adam
Base learning rate (LR)	1×10^{-4}
Minimum learning rate (MIN_LR)	1×10^{-6}
Warmup steps (WARMUP_STEPS)	16
Weight decay (WEIGHT_DECAY)	1×10^{-4}
Gradient clipping (GRAD_CLIP)	1.0
Loss function (LOSS)	SILog (masked over valid pixels)
SILog mixing weight (SILOG_LAMBDA)	0.5
Numerical stability epsilon (EPS)	1×10^{-4}
Prediction activation (PRED_ACT)	Softplus
Training resolution (OUT_W $\times$ OUT_H)	512×512

begins on a chunk only after rendering has fully produced the RGB image, depth target, and required metadata for every sample in that chunk. This prevents partial reads and ensures that the learner observes consistent supervision.

3.5.4 ResUNet18 Training Implementation in LibTorch

Depth prediction is implemented and trained in LibTorch using a ResUNet18 architecture. The model follows an encoder–decoder design in which the encoder is a ResNet-18 backbone and the decoder progressively upsamples feature maps while fusing skip connections from corresponding encoder stages. The encoder provides multi-scale semantic features, and the skip connections preserve high-frequency spatial detail required for sharp boundaries and fine structures in the depth map. The final prediction is a dense, single-channel, camera-centric depth map aligned to the 512×512 input.

Training uses Adam and is executed for 30 epochs. Increasing the epoch count is motivated by the need to reduce undertraining artifacts such as holes, discontinuities, and unstable depth scale, which can persist when the network has not sufficiently absorbed the geometric variability across categories and views. Validation and test evaluation are performed on fixed 10% partitions to provide consistent generalization tracking across epochs.

3.5.5 3D Reconstruction with LibTorch CUDA Tensors

The reconstruction stage converts the predicted camera-centric depth map into an explicit 3D surface representation. Although reconstruction is computationally heavy due to dense per-pixel and per-quad processing, the underlying operations are deterministic and strongly data-parallel. For this reason, the implementation leverages LibTorch CUDA tensors to execute the dominant geometric computations on the GPU, minimizing additional engineering overhead while preserving practical runtimes under limited hardware resources. The experiments are conducted with

CUDA 12.0.
For an image of size $H \times W$, vertex generation scales as $O(HW)$ because each pixel is independently unprojected into a 3D point. Triangle mesh construction over the image lattice scales as $O((H-1)(W-1))$ because each quad in the 2D grid is tested independently for validity and may contribute up to two triangles. In addition to unprojection, the reconstruction includes validity masking (to exclude non-finite depth, background depth, or masked-out regions) and a depth discontinuity rejection rule to avoid forming triangles across occlusion boundaries. These stages are expressed primarily as GPU tensor operations (indexing, broadcasting, and element-wise comparisons).

Vertex Grid Generation on the GPU

Let $\hat{Z} \in R^{H \times W}$ denote the predicted depth map, and let $m \in \{0,1\}^{H \times W}$ be a validity mask indicating whether a pixel should be reconstructed. Using the pinhole camera model, each pixel (u, v) is lifted into a 3D vertex in the camera coordinate frame:

$$\mathbf{P}_{\text{cam}}(u,v) = \hat{Z}(u,v) \cdot \mathbf{K}^{-1} \begin{bmatrix} u \\ v \\ 1 \end{bmatrix}. \tag{3.16}$$

The full vertex field is maintained as a dense $H \times W$ grid on the GPU. Invalid vertices are written as a sentinel value (e.g., NaN) so they can be excluded from later steps using boolean masks:

$$\mathbf{V}(u,v) = \begin{cases} \mathbf{P}_{\text{cam}}(u,v), & m(u,v) = 1 \ \wedge \ \hat{Z}(u,v) \text{ finite}, \\ \text{NaN}, & \text{otherwise.} \end{cases} \tag{3.17}$$

Grid-Based Triangle Mesh Without Vertex Compaction

To reduce implementation complexity, the mesh is constructed without compacting the vertex list. The index of each vertex is defined by its original image-grid position:

$$\text{idx}(u,v) = vW + u, \tag{3.18}$$

where $u \in \{0, \ldots, W-1\}$ and $v \in \{0, \ldots, H-1\}$. This indexing rule allows faces to be emitted directly in terms of the original raster layout, avoiding stream compaction and index remapping.
For each image quad with top-left corner (u, v), define the four vertex indices:

$$i_{00} = \text{idx}(u,v), \quad i_{10} = \text{idx}(u+1,v), \quad i_{01} = \text{idx}(u,v+1), \quad i_{11} = \text{idx}(u+1,v+1). \tag{3.19}$$

If the four vertices are valid, the quad can be triangulated into two faces using the standard lattice connectivity:

$$\Delta_1 = (i_{00}, i_{10}, i_{01}), \qquad \Delta_2 = (i_{10}, i_{11}, i_{01}). \tag{3.20}$$

Validity and Depth Discontinuity Constraints

A naive lattice triangulation can incorrectly connect foreground and background across depth jumps. To prevent this, a discontinuity rule is enforced prior to face

emission. Let $\mathbf{v}_{00}, \mathbf{v}_{10}, \mathbf{v}_{01}, \mathbf{v}_{11}$ denote the corresponding 3D vertices from the dense grid $\mathbf{V}$. First, faces are emitted only when all participating vertices are valid together with a finite-depth requirement:

$$\mathcal{M}_{\text{quad}}(u, v) = m(u, v)\, m(u+1, v)\, m(u, v+1)\, m(u+1, v+1), \tag{3.21}$$

Second, discontinuity rejection is enforced using an edge-length threshold. Let

$$d(\mathbf{a}, \mathbf{b}) = \|\mathbf{a} - \mathbf{b}\|_2. \tag{3.22}$$

A triangle is emitted only if all of its edges are shorter than a threshold τ_{edge}:

$$d(\mathbf{v}_p, \mathbf{v}_q) < \tau_{\text{edge}} \quad \text{for all edges } (p, q) \text{ in the triangle.} \tag{3.23}$$

Equivalently, per-quad boolean masks can be computed for Δ_1 and Δ_2 and used to gate face emission. These masks are computed on the GPU using LibTorch tensor operations (neighbor indexing and element-wise comparisons), which is efficient because each quad can be processed independently.

Export Strategy Without Remapping

Because vertices are not compacted, the exported vertex array contains HW entries, including invalid vertices written as NaN (or a fixed sentinel). Faces are emitted only for triangles whose validity and discontinuity constraints pass, ensuring that no face references invalid vertices. This design avoids complicated vertex compaction and index remapping while still producing a triangle mesh suitable for visualization and qualitative inspection. In practice, many mesh viewers and downstream tools accept meshes where invalid vertices exist but are never referenced by any face. If required for strict cleanliness, a later optional pass can compact vertices and remap indices, but this is not necessary for the core reconstruction demonstrated in this thesis.

Overall, this approach achieves GPU-accelerated reconstruction with minimal additional implementation complexity, because vertex generation, validity masks, and triangle eligibility masks are expressed directly as tensor operations within the LibTorch CUDA execution model.

Chapter 4

Results and Analysis

An empirical evaluation of the proposed depth-centric 2D–to–3D reconstruction system, focusing on measured system behavior under sustained workload and quantitative depth estimation performance across training batches. All results are derived from runtime logs, evaluation traces, and metric histories collected during full training runs, rather than from isolated or post-hoc experiments.

The evaluation points out the dual nature of the pipeline. First of all, it is a flow of work in the area of systems shown to work constantly to have limited hardware in order to make its practical. So, validity involves the stable behavior of utilization and memory as a function of time. Second is depth estimation model, that is based on learning and its accuracy and numerical. Stability need to be maintained over heterogeneous shards, over different scenery difficulty For this reason, results are provided on system level and model level, their interaction by implicit reflecting behaviour (cor correlated run time and metrics).

System level analysis reporting GPU utilization GPU Memory Usage CPU utilization ,and and host RSS memory over time. These time traces are used to tell if the training loop keep up the saturating of GPU going, and prevent data starving, and is stable and not increased on the long term memory. The plots give the opportunity to point out the synchronisation overheads as well as batch or shard transition effects & bottlenecks which could prevent of long duration executing these.

Model level evaluation is based on common measurements for depth estimation for the following: Mean, absolute error (MAE) Root, mean square, error (RMSE) absolute (AbsRel) and accuracy under threshold (δ_1, δ_2, δ_3). Metrics are determined for each batch-level and are displayed in both raw and smoothed forms. The raw curves contain the variability on the level of shard while the smoothed curves emphasize the long term trends reducing the effect of transient difficult batches.

Importantly, the results being reported are taking pains staying out the full variability in the training, also for some batches higher accuracy is not so high (temporarily low). This variability is not considered as noise to be removed, but as an indication for robustness operating under realistic conditions with change in shard composition & difficulty over time. Taken all together the results produces and assessment of if the system fill its core design objectives: stable and long duration execution, efficient use of computational resources and an estimation execution performance in constant depth, regardless of the data of the stream.

4.1 Model Performance Evaluation

4.1.1 Root Mean Square Error (RMSE)

The RMSE curve has perhaps a little too much ”batch-to-batch” variability, with raw values spanning a broad range. This variability is expected because of heterogeneity in the scene geometry, objects scale & contradict depth discontinuities between batches.

When smoothing using a moving average window, RMSE will converge to a steady state and a persistent operating band without no down collapsing or upward divergence of the band. This suggests that some individual batches are difficult, but the model is not overfitting on individual shards, but overall.

The fact that there are isolated spikes of RMSE, and this is immediately followed by recovery, suggests too sensitive than usual towards some configuration of scenes rather than systematic failure. The so far delivered findings show that such spikes are not transferred to the following batches, proving that the explicit representations inside the model are stable from its train iteration to other.

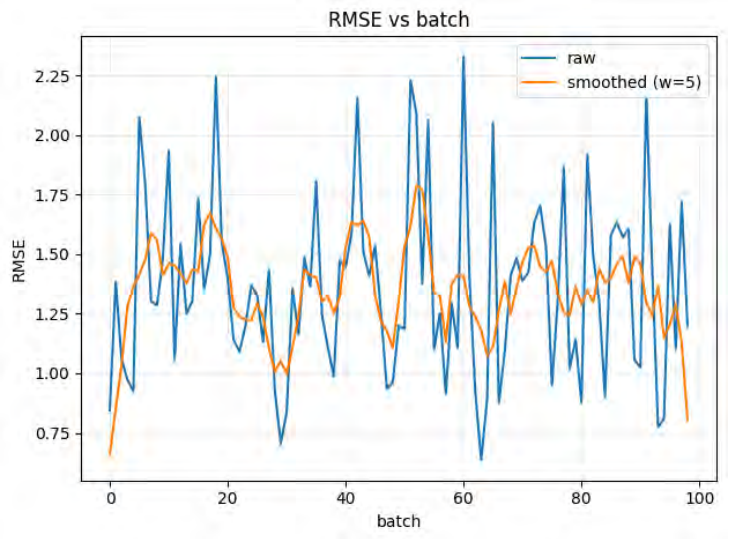


Figure 4.1: Root Mean Square Error (RMSE) across training batches.

4.1.2 Mean Absolute Error (MAE)

MAE follow simial form of RMSE but with reduced sensitivity to extremes. outliers. The raw MAE signal is fairly variable, with some peaks corresponding to batches with high absolute deviations of depth.

The curve of the smoothed MAE is always bounded. This implies that the model maintains constant absolute depth accuracy with time, and the absence of sustained upward drift is to check if the model doesn't get bias/scale error during the training process. progresses.

Compared to RMSE More smooth MAE behavior implies an implication that big errors are big errors. more frequent & localised re-enforces conclusion that most predictions remain No discrepancy in the absolute values of the measures is more than reasonable error margins.

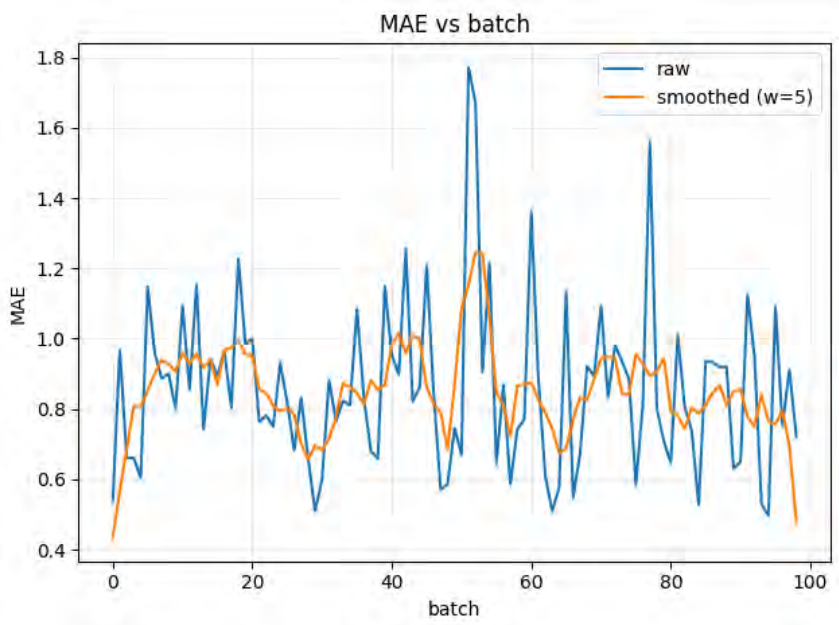


Figure 4.2: Mean Absolute Error (MAE) across training batches.

4.1.3 Absolute Relative Error (AbsRel)

AbsRel has sharper peaks than MAE - is more sensitive to the relative scale mismatches. Parts are created for scenes with high AbsRel values where the predicted depth is accounting to be deviation from ground truth.

However the smoothed trend of AbsRel is stable in the full range of the training, exhibiting no deterioration in the scale collapse or divergence. This stability is described by the particularly important in monocular based depth estimation where scale ambiguity is very well known failure mode.

The aspect of this model that the recovery of AbsRel after spikes might be that the model does not lock-in to wrong scale hypotheses and adapts instead in a variety of distribution of scenes.

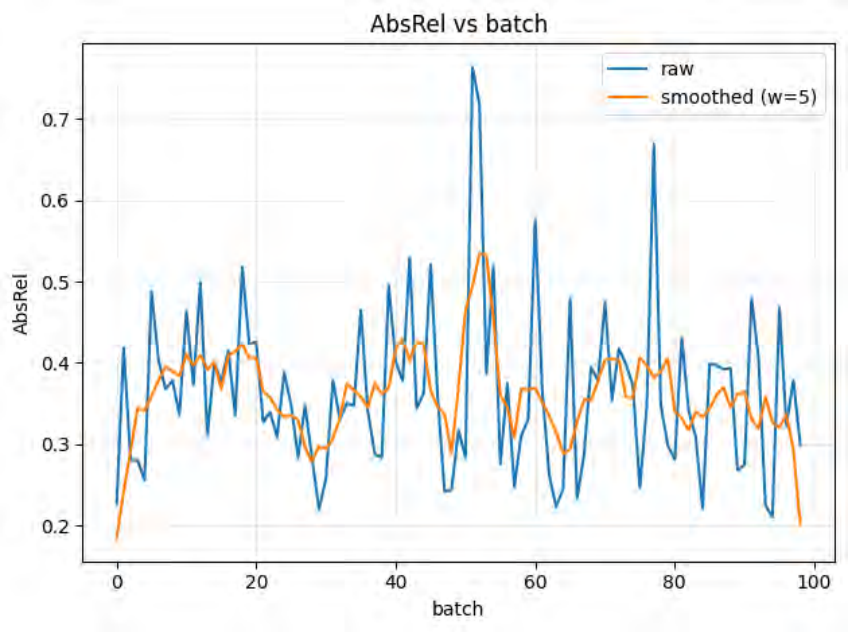


Figure 4.3: Absolute Relative Error (AbsRel) across batches.

4.1.4 δ_1 Accuracy

The δ_1 accuracy curve has shown a good deal of level of volatility on the batch level including sharp drops on really tough batches. These drops are indicative of cases where strict accuracy threshold is not fulfilled for great proportion of pixels.

These failings are however, only temporary. The smoothed curve forδ_1 is stable and the accuracy again increase back with the next batches. This is a sign of a robust and not fragile model: The model sometimes has a hard time with difficult scenes but doesn't work in non-catastrophic manner and not degrade.

This is a pattern that can be seen also in other different models that have been trained using diverse data, i.e. strict thresholds, which can expose difficult corner cases without losing the overall reliability.

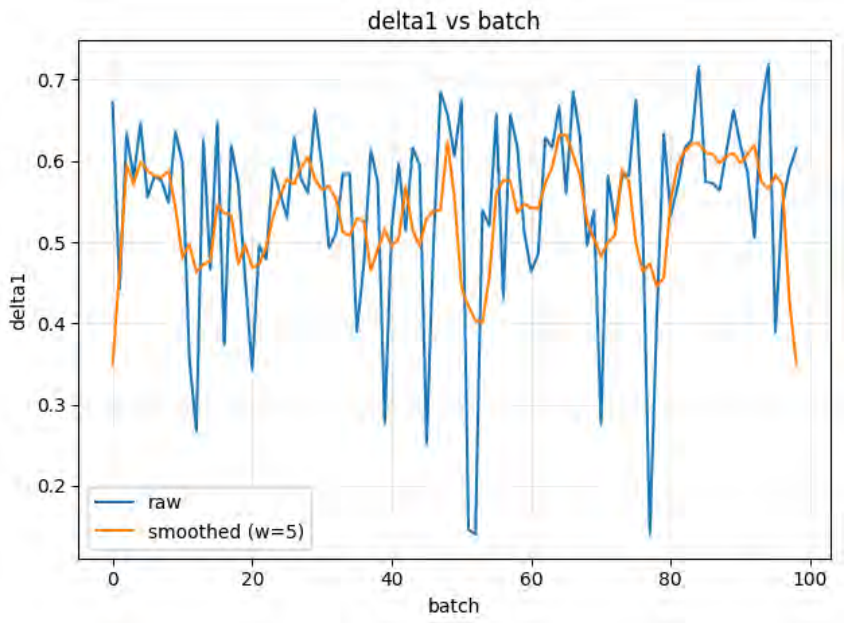


Figure 4.4: δ_1 accuracy across training batches.

4.1.5 δ_2 and δ_3 Accuracy

δ_2 and $\delta_3$accuracy curves depict higher stability along with lower variance progressively. δ_2 is bad with some dips but δ_3 is close to saturation in most all batches.

The good performance in terms of δ_3 indicates that extreme depth prediction failures are rare. Even when there is not a good level of accuracy, the predictions are on a reasonable error level, with geometric plausibility for reconstruction down stream.

The combined behavior of δ_1, δ_2 , and δ_3 , errors are found to be predominantly moderate instead of being catastrophic which is a critical aspect for practical 3D reconstruction.

Figure 4.5: δ_2 accuracy across batches.

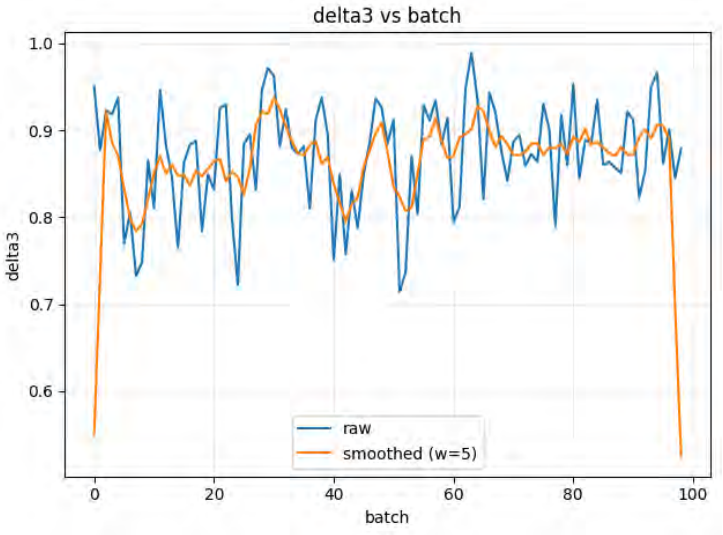


Figure 4.6: δ_3 accuracy across batches.

4.1.6 Training Stabilty Summary

In all its metrics, the defining feature of the observed behaviour is bounded variance with high speed of recuperation. No metrics shows collapse, runaway growth or long term degradation. This confirms, the training process is running in a stable optimization regime.

Table 4.1: Overall statistical summary of depth estimation performance across all evaluated batches.

Metric	Mean	Std. Dev.	Variance	Min	Max
MAE	0.859	0.239	0.057	0.496	1.773
RMSE	1.360	0.379	0.144	0.426	2.330
AbsRel	0.365	0.103	0.011	0.197	0.764
δ_1	0.543	0.122	0.015	0.140	0.867
δ_2	0.779	0.089	0.008	0.408	0.918
δ_3	0.872	0.061	0.004	0.714	0.947

The observed variability reflects dataset complexity rather than numerical instability or architectural mismatch. As a result, the model achieves a reliable trade-off between expressiveness and stability, consistent with the design goals of a low-overhead CNN-based depth estimation system.

4.1.7 Comparative Analysis of Depth Prediction Model

To contextualize the quantitative performance of the proposed depth prediction module, we compare ResUNet18 against a set of representative monocular depth estimation methods reported in the literature. The comparison is presented using standard evaluation criteria: threshold accuracies (δ_1, δ_2, δ_3) and error measures (REL and RMS). Threshold metrics are reported such that higher values indicate improved accuracy, whereas lower values are preferable for REL and RMS. This comparative view is intended to position the proposed approach relative to established baselines and state-of-the-art systems, while highlighting the design trade-offs adopted in this thesis.

Table 4.2 shows that ResUNet18 does not reach the highest reported accuracy achieved by recent specialized architectures (e.g., AdaBins), particularly in terms of REL and δ-threshold performance. However, the objective of this thesis is not solely to maximize benchmark accuracy; instead, the model is selected and evaluated as a component of a complete depth-centric 2D to 3D reconstruction pipeline under constrained hardware conditions. Within this scope, ResUNet18 provides stable training behavior and strong throughput, while producing depth maps of sufficient to support the downstream reconstruction stage. Consequently, the proposed design represents a practical trade off between accuracy and computational efficiency, aligned with the system-level goals of sustained execution and reliable 3D reconstruction from predicted depth.

Table 4.2: Metrics comparison against representative depth prediction methods.

Method (paper)	$\delta_1 \uparrow$	$\delta_2 \uparrow$	$\delta_3 \uparrow$	REL $\downarrow$	RMSE $\downarrow$
Multi-Scale CNN (2014)	0.769	0.95	0.988	0.158	0.641
FCRN (2016)	0.811	0.953	0.988	0.127	0.573
Attention Guided (2018)	0.841	0.966	0.991	0.127	0.555
Deep Ordinal (2018)	0.828	0.965	0.992	0.115	0.509
SharpNet (2019)	0.836	0.966	0.993	0.139	0.502
Revisiting (2019)	0.866	0.975	0.993	0.115	0.53
Structure-aware (2019)	0.878	0.977	0.994	0.111	0.514
Virtual Normal (2019)	0.875	0.976	0.994	0.108	0.416
BTS (2019)	0.885	0.978	0.994	0.11	0.392
DAV (2020)	0.882	0.98	0.996	0.108	0.412
AdaBins	0.903	0.984	0.997	0.103	0.364
ResUNet18 (Ours)	0.867	0.918	0.947	0.197	0.426

4.1.8 Epoch Time and Training-Time Comparison

Training-time behavior is analyzed across smaller and larger network architectures under an identical training pipeline and the same hardware configuration. This comparison provides a system-level perspective on computational cost and throughput.

Table 4.3: Training throughput and resource utilization comparison across architectures under the same pipeline and hardware.

Model	Images processed per Epoch	Average Epoch Time (mins)	GPU usage (%)
UNet	1200	45.37	98
ResNet18	1200	63.45	98
ResUNet18	1200	54.28	97

The results in Table 4.3 indicate that the proposed ResUNet18 model exhibits competitive training throughput relative to both a smaller baseline (UNet) and a larger encoder-dominant backbone (ResNet18), while sustaining consistently high GPU utilization. For ResUNet18, the measured average epoch time is 54.28 minutes for 1200 images, corresponding to an average processing time of 2.714 seconds per image. All measurements were collected on the same compute platform consisting of an NVIDIA RTX 3050 GPU and an Intel Core i7 CPU (8 cores). GPU usage denotes the average GPU compute utilization (%) measured during training.

4.2 System Level Performance Analysis

4.2.1 GPU Utilisation

The GPU utilization trace depicts utilization for very high throughout the execution window, where the utilisation signal is periodically almost reaching the saturation of levels for 80 percent of the time while running. This harmonius to the fact that the computational The workload forward and backward are dominated by the

convolution operations passes–if constantly being supplied to the GPU without runtime blanking.

The sharp and periodical dropping of utilisation is associated with discrete synchronisation occurring events, as opposed to of very systemic inefficiencies. These drops are synchronized with each other (batch-wise) towards the time Look, transitions shard boundaries, logging or checkpoint operations, however, are important: The utilization recovers back after every drop - implying than pipeline Is true because message queue is empty. CPU-side processing does not stop as data is not available. This behavior is confirmed that the producer-consumer design can overlap data preparation and GPU execution.

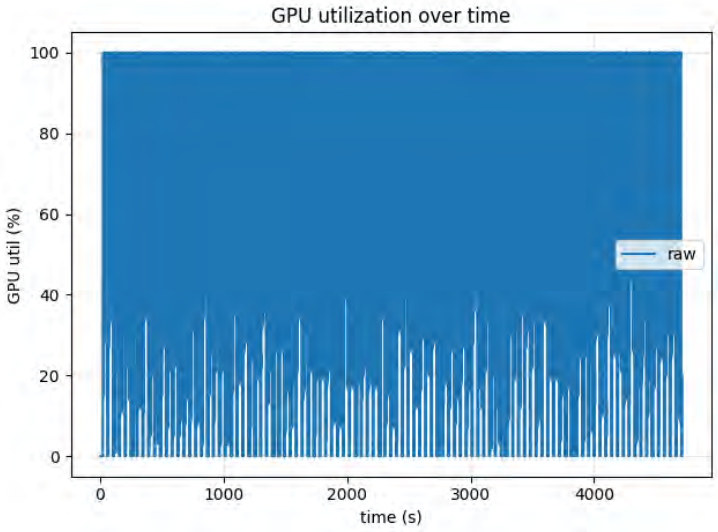


Figure 4.7: GPU utilization over time during training.

As opposed to the I/O blocking/ too much preprocessing in the host-side of the pipelines, no steady low utilization for a period of time from the plateaus is found. The absence of such plateaus is used if it is found that GPU compute sat is being dominated not so much by the GPU compute, but by the model itself So much more, though, to be held back due to outside pipeline capabilities.

GPU Memory Usage

Throughout the runtime, GPU memory usage remains extremely tightly constrained, ranging from nearly zero allocation during batch transitions to a steady peak during active computation. The peak in the footprint of the memory is the same for batches, indicating that the size of the tensor allocation does not increase with time.

This behaviour adds heavy proof to the fact that memory is recursively being reused and that the intermediate tensors used are released every iteration. No upward drift and creeping increase in memory usage is noticed, ruling out memory leaks or working on the fragmentation. The sawtooth pattern is repeated, which is the result of deliberate allocation-deallocation cycles due to batch-level execution as opposed to uncontrolled memory churn.

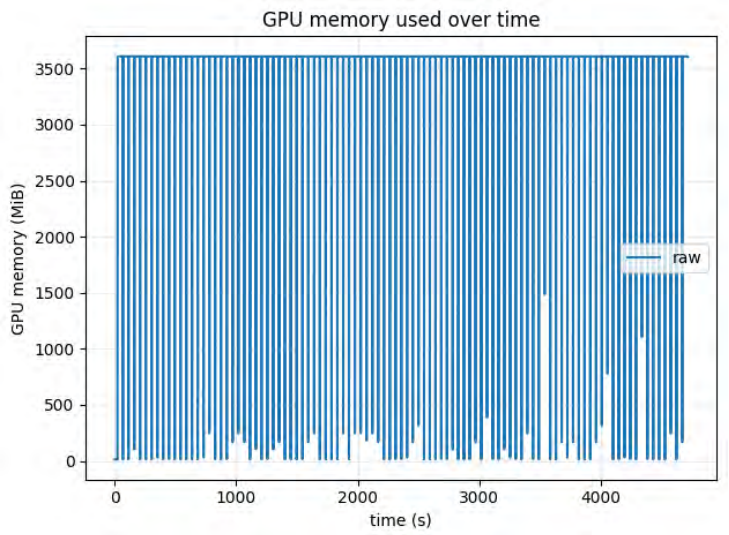


Figure 4.8: GPU memory usage over time.

Crucially the fact that the peak memory usage was stable proves that the system can support training runs with a long time duration without the need of manual intervention/restarts which is a growing example of less well disciplined GPUs pipelines normal failure mode.

4.2.2 CPU and I/O Behavior

Pipeline Balance

Aggregate CPU utilization, since during the whole process of training the computer, CPU utilization is in a steady working position, i.e., the second complementary is not monotonic, it only the fluctuate arounda stable average. growth or collapse. This indicates that tasks on the CPU side are not getting utilized such as data set shard orchestration, preprocessing ,and logging, syncing up the tasks getting on and off, are not getting utilized. nor overwhelmed.

The periodic variation of the CPU usage are cyclic in nature of batches level operations instead of unsolicited contention Also of note the CPU utilization does not have kept peaks sometime as blank GPU confirms CPU-side. throttling of the throughput of the GPU according to execution. The architectural decision to separate rendering, caching, and training in coordinated pipelines stages instead of running them one after the other.

This balance validates the architectural decision to decouple rendering, caching, and training into coordinated pipeline stages rather than executing them sequentially.

Host Memory (RSS) Usage

The RSS memory trace exhibits bounded fluctuations throughout execution, with repeated rises and falls corresponding to buffer allocation and cleanup events. While transient peaks are visible, these peaks do not accumulate over time.

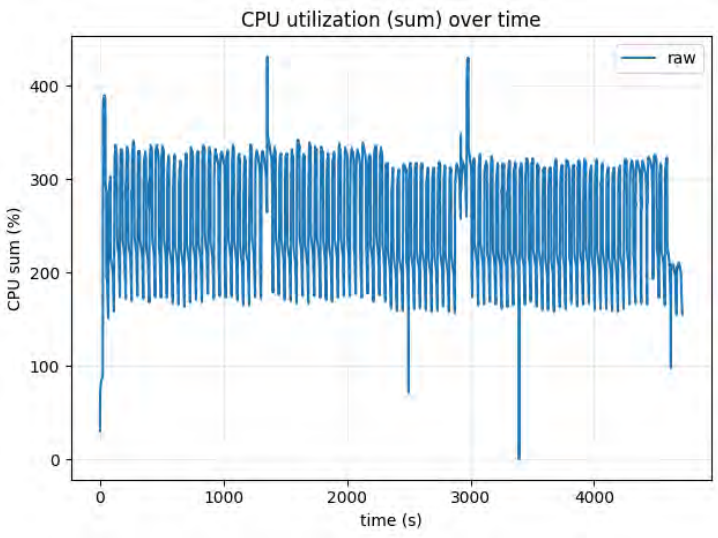


Figure 4.9: Aggregate CPU utilization over time.

The assumption that RSS should be a monotonic increase is confirmed. Host-side memory reclaiming occurs within a time following each of the batches or shards, and the cached data structures do not last Proceed beyond their life span of design. Occasional sharp dips in the RSS use indicate an explicit cleanup or scope exit points, as well as supporting the ending of the memory. Management is intentional (rather than accidental to nature).

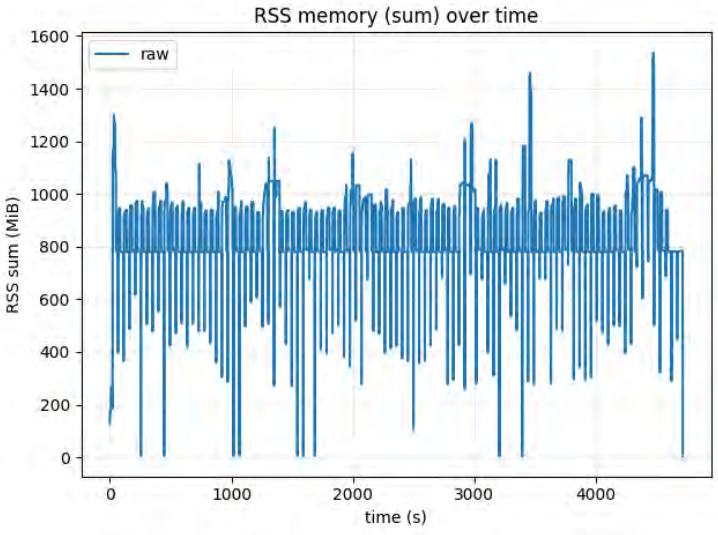


Figure 4.10: Host RSS memory usage over time.

From a systems perspective, such behavior is proof of the robustness of the pipeline against the pressure of the long-term memory, it is able to work for a long time, without consuming the host memory resources.

Predicted Depth Visualization

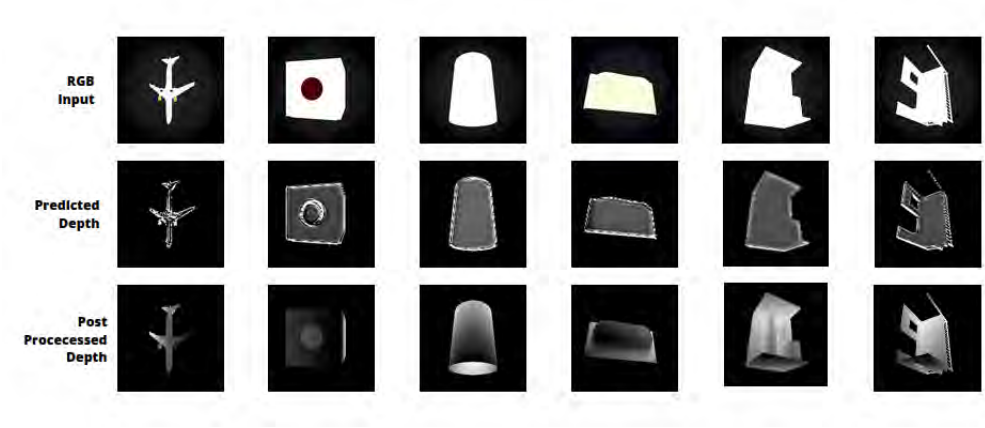


Figure 4.11: RGB input to predicted depth and post processed depth

The above Predicted Depth Visualization outcomes are very promising, and the visualization perfectly captures different objects with clear shapes and depth information. The depth here is coming from the RGB input, and it is showing various objects with promising outcomes and clear geometrical shapes. Overall, it is a very successful outcome, and it shows my pipeline for creating depth value Z is working very well.

PLY Plot Visualization

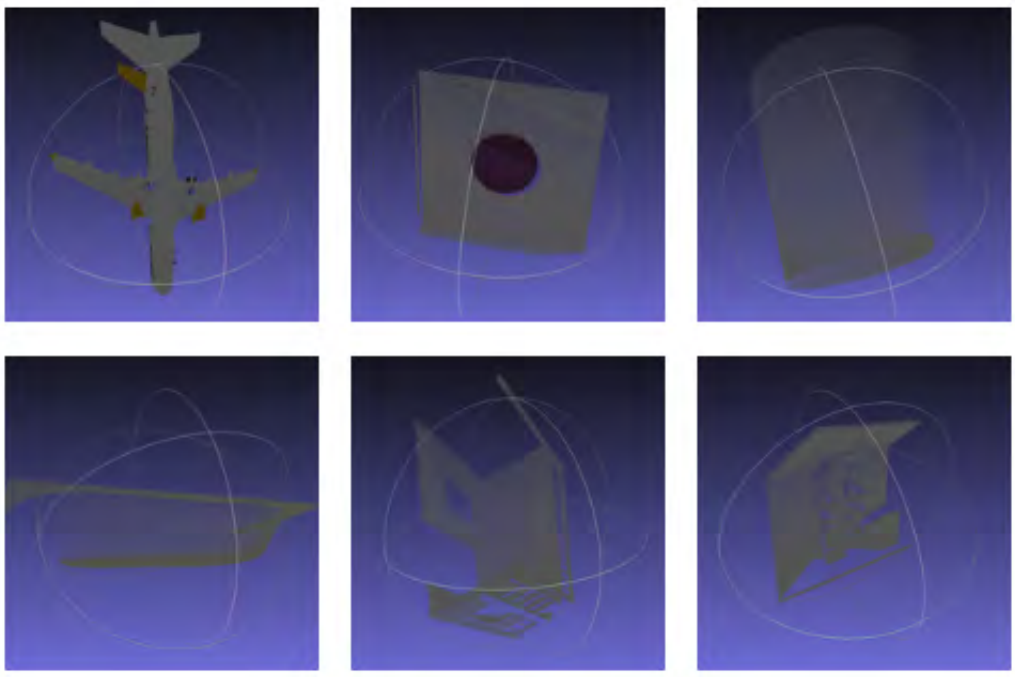


Figure 4.12: PLY Plot Visualization

Looking at the PLY Plot Visualization, the outcomes are very promising. The visualization perfectly captures different objects like planes, geometrical shapes,

etc., all rendered within the spherical frames. The conversion from RGB input to predicted depth (Z-Depth) and later processing the depth is coming along very well and showing very clear geometrical representation of the various objects. Overall, the PLY plot visualization illustration proves that the depth estimation pipeline is performing effectively and producing better 3D representations.

Chapter 5

Conclusion

This thesis investigated the practical feasibility of depth-centric 2D-to-3D reconstruction under realistic hardware constraints, motivated by the fundamental ambiguity of recovering 3D structure from a single 2D projection and the increasing computational cost of modern reconstruction pipelines. Rather than relying on strict multi-view acquisition assumptions, the work adopted a monocular depth estimation approach and treated the overall workflow as a systems problem: sustaining training and reconstruction when GPU memory, host memory, and I/O bandwidth are limited.

The primary outcome is an end-to-end pipeline that couples monocular depth prediction with explicit 3D reconstruction. A ResUNet-style network with a ResNet18 encoder was trained in LibTorch to infer dense depth from single RGB inputs, producing depth maps that can be lifted into 3D via camera back-projection. This design follows the progression from image representations and depth inference to reconstruction, where depth serves as a practical intermediate representation for generating point clouds and grid-based meshes. The outcomes of the reported qualitative results reveal that the quality of depth prediction is consistent throughout batches of training under the typical depth estimation metrics which led to a consistent learning behavior during sustained runs.

A central contribution of this thesis is the emphasis on scalable execution on low-overhead computers, achieved through an SSD-backed producer–consumer workflow. By decoupling rendering from training through sharded streaming, the pipeline prevents data starvation and avoids unbounded growth in storage or memory usage.The observed runtime shows that during training, bounded GPU and host memory behavior across an extended run maintains sustained GPU utilization, while demonstrating that pipeline-level engineering is crucial when datasets are being generated by rendering.

On the reconstruction side, the thesis demonstrated that CUDA-enabled depth-to-geometry conversion makes the 3D stage computationally practical at image-grid scale. Dense back-projection and point/mesh generation are inherently parallel, and GPU execution reduces CPU bottlenecks that would otherwise dominate runtime as resolution increases. In this work, CUDA acceleration improves throughput and reconstruction efficiency while preserving the underlying geometric formulation and interpretability of the reconstruction process.

These outcomes directly address the research questions by showing that the end-to-end depth-centric pipeline can operate stably on constrained hardware, that the producer–consumer sharded workflow sustains training without data starvation under bounded storage and memory, and that CUDA-accelerated reconstruction reduces geometry-generation overhead without degrading the usability of the resulting point clouds and meshes.

One limitation is that monocular depth estimation remains inherently ambiguous, and reconstruction quality is sensitive to depth discontinuities and invalid regions, which can lead to gaps or artifacts and may limit generalization beyond the rendered training distribution.

Overall, the findings show that a depth-centric 2D-to-3D pipeline can be made operationally viable on constrained hardware when model design, data movement, and reconstruction are treated as a unified system. This thesis contributes an end-to-end depth-to-3D workflow suitable for sustained operation, a bounded producer–consumer streaming architecture for rendered datasets, and CUDA-accelerated reconstruction for efficient geometry generation. Future work may extend this foundation by exploring multi-view fusion for improved completeness, incorporating stronger depth supervision or domain adaptation for better generalization, and adding surface reconstruction and filtering methods to improve mesh quality while preserving the low-overhead design goals.

Bibliography

[1] M. Subbarao and G. Surya, *Depth from defocus: A real aperture imaging approach*, https://ieeexplore.ieee.org/document/1163711, CVPR, 1994.

[2] B. Curless and M. Levoy, "A volumetric method for building complex models from range images," in *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH 1996, New Orleans, LA, USA, August 4–9, 1996*, J. Fujii, Ed., ACM, 1996, pp. 303–312. DOI: 10.1145/237170.237269 [Online]. Available: https://doi.org/10.1145/237170.237269

[3] L. S. Blackford et al., "An updated set of basic linear algebra subprograms (BLAS)," *ACM Transactions on Mathematical Software*, vol. 28, no. 2, pp. 135–151, Jun. 2002. DOI: 10.1145/567806.567807

[4] R. Hartley and A. Zisserman, *Multiple view geometry in computer vision*. Cambridge university press, 2003.

[5] R. I. Hartley and A. Zisserman, *Multiple View Geometry in Computer Vision*, Second. Cambridge University Press, 2004, ISBN: 0521540518.

[6] S. M. Seitz, B. Curless, J. Diebel, D. Scharstein, and R. Szeliski, *A comparison and evaluation of multi-view stereo reconstruction algorithms*, https://ieeexplore.ieee.org/document/4359315, CVPR, 2006.

[7] P. Merrell et al., "Real-time visibility-based fusion of depth maps," in *2007 IEEE 11th International Conference on Computer Vision (ICCV)*, 2007. [Online]. Available: https://mordohai.github.io/public/Merrell_DepthMapFusion07.pdf

[8] R. A. Newcombe et al., "Kinectfusion: Real-time dense surface mapping and tracking," in *10th IEEE International Symposium on Mixed and Augmented Reality, ISMAR 2011, Basel, Switzerland, October 26–29, 2011*, IEEE Computer Society, 2011, pp. 127–136. DOI: 10.1109/ISMAR.2011.6092378 [Online]. Available: https://doi.org/10.1109/ISMAR.2011.6092378

[9] R. B. Rusu and S. Cousins, "3d is here: Point cloud library (pcl)," in *2011 IEEE International Conference on Robotics and Automation (ICRA)*, 2011. DOI: 10.1109/ICRA.2011.5980567 [Online]. Available: https://pointclouds.org/assets/pdf/pcl_icra2011.pdf

[10] S. Chetlur et al., "Cudnn: Efficient primitives for deep learning," *arXiv preprint arXiv:1410.0759*, 2014. arXiv: 1410.0759 [cs.NE]. [Online]. Available: https://arxiv.org/abs/1410.0759

[11] D. Eigen, C. Puhrsch, and R. Fergus, "Depth map prediction from a single image using a multi-scale deep network," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2014.

[12] I. Goodfellow et al., "Generative adversarial nets," *Advances in neural information processing systems*, vol. 27, 2014.

[13] M. Mirza, "Conditional generative adversarial nets," *arXiv preprint arXiv:1411.1784*, 2014.

[14] A. X. Chang et al., "Shapenet: An information-rich 3d model repository," *CoRR*, vol. abs/1512.03012, 2015. arXiv: 1512.03012. [Online]. Available: http://arxiv.org/abs/1512.03012

[15] A. Radford, "Unsupervised representation learning with deep convolutional generative adversarial networks," *arXiv preprint arXiv:1511.06434*, 2015.

[16] O. Ronneberger, P. Fischer, and T. Brox, "U-net: Convolutional networks for biomedical image segmentation," in *Medical image computing and computer-assisted intervention–MICCAI 2015: 18th international conference, Munich, Germany, October 5-9, 2015, proceedings, part III 18*, Springer, 2015, pp. 234–241.

[17] Ö. Çiçek, A. Abdulkadir, S. S. Lienkamp, T. Brox, and O. Ronneberger, "3d u-net: Learning dense volumetric segmentation from sparse annotation," in *Medical Image Computing and Computer-Assisted Intervention–MICCAI 2016: 19th International Conference, Athens, Greece, October 17-21, 2016, Proceedings, Part II 19*, Springer, 2016, pp. 424–432.

[18] Y. Furukawa and C. Hernandez, *Multi-view stereo: A tutorial*, https://www.cv-foundation.org/openaccess/content_cvpr_2016/html/Furukawa_Multi-View_Stereo_CVPR_2016_paper.html, CVPR Tutorial, 2016.

[19] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Jun. 2016.

[20] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," *arXiv preprint arXiv:1609.02907*, 2016.

[21] I. Laina, C. Rupprecht, V. Belagiannis, F. Tombari, and N. Navab, "Deeper depth prediction with fully convolutional residual networks," in *2016 Fourth international conference on 3D vision (3DV)*, IEEE, 2016, pp. 239–248.

[22] J. L. Schönberger and J. M. Frahm, *Colmap: Structure-from-motion and multi-view stereo*, https://demuc.de/colmap/, Software package, 2016.

[23] J. L. Schönberger and J.-M. Frahm, "Structure-from-motion revisited," in *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[24] J. Xie, R. Girshick, and A. Farhadi, "Deep3d: Fully automatic 2d-to-3d video conversion with deep convolutional neural networks," in *European conference on computer vision*, Springer, 2016, pp. 842–857.

[25] T. Kalaiselvi, P. Sriramakrishnan, and K. Somasundaram, "Survey of using gpu cuda programming model in medical image analysis," *Informatics in Medicine Unlocked*, vol. 9, pp. 133–144, 2017.

[26] J. Lee, H. Jung, Y. Kim, and K. Sohn, "Automatic 2d-to-3d conversion using multi-scale deep neural network," in *2017 IEEE International Conference on Image Processing (ICIP)*, 2017, pp. 730–734. DOI: 10.1109/ICIP.2017.8296377

[27] G. Riegler, A. O. Ulusoy, and A. Geiger, "Octnet: Learning deep 3d representations at high resolutions," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017, pp. 6620–6629. DOI: 10.1109/CVPR.2017.701

[28] P.-S. Wang, Y. Liu, Y.-X. Guo, C.-Y. Sun, and X. Tong, "O-cnn: Octree-based convolutional neural networks for 3d shape analysis," *ACM Transactions on Graphics*, vol. 36, no. 4, 72:1–72:11, 2017. DOI: 10.1145/3072959.3073608

[29] Z. Zhang, Q. Liu, and Y. Wang, "Road extraction by deep residual u-net," *arXiv preprint arXiv:1711.10684*, 2017.

[30] "2018 ieee 61st international midwest symposium on circuits and systems (mwscas)," in *2018 IEEE 61st International Midwest Symposium on Circuits and Systems (MWSCAS)*, Piscataway, NJ, USA: IEEE, Aug. 2018. DOI: 10.1109/MWSCAS.2018.8624094 [Online]. Available: https://ieeexplore.ieee.org/xpl/conhome/8624094/proceeding

[31] "2018 ieee 61st international midwest symposium on circuits and systems (mwscas)," in *2018 IEEE 61st International Midwest Symposium on Circuits and Systems (MWSCAS)*, Piscataway, NJ, USA: IEEE, Aug. 2018. DOI: 10.1109/MWSCAS.2018.00001 [Online]. Available: https://ieeexplore.ieee.org/xpl/conhome/8624094/proceeding

[32] H. Fu, M. Gong, C. Wang, K. Batmanghelich, and D. Tao, "Deep ordinal regression network for monocular depth estimation," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018. [Online]. Available: https://openaccess.thecvf.com/content_cvpr_2018/papers/Fu_Deep_Ordinal_Regression_CVPR_2018_paper.pdf

[33] D. B. Kirk and W. W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach*, 3rd. Morgan Kaufmann, 2018.

[34] L. A. Lopez, K. Auchettl, T. Linden, A. D. Bolatto, T. A. Thompson, and E. Ramirez-Ruiz, "Evidence for cosmic-ray escape in the small magellanic cloud using fermi gamma-rays," *The Astrophysical Journal*, vol. 867, no. 2, p. 112, Nov. 2018. DOI: 10.3847/1538-4357/aae0f8 arXiv: 1807.06595 [astro-ph.HE].

[35] Z. Feng, Z. Chao, Y. Huamin, and D. Yuying, "Research on fully automatic 2d to 3d method based on deep learning," in *2019 IEEE 2nd International Conference on Automation, Electronics and Electrical Engineering (AUTEEE)*, 2019, pp. 538–541. DOI: 10.1109/AUTEEE48671.2019.9033402

[36] C. Godard, O. Mac Aodha, M. Firman, and G. J. Brostow, "Digging into self-supervised monocular depth estimation," in *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, 2019. [Online]. Available: https://openaccess.thecvf.com/content_ICCV_2019/papers/Godard_Digging_Into_Self-Supervised_Monocular_Depth_Estimation_ICCV_2019_paper.pdf

[37] A. Paszke et al., "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems*, vol. 32, 2019, pp. 8024–8035. [Online]. Available: https://proceedings.neurips.cc/paper/2019/hash/bdbca288fee7f92f2bfa9f7012727740-Abstract.html

[38] D. A. Rockenbach et al., "Stream processing on multi-cores with gpus: Parallel programming models' challenges," in *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, IEEE, 2019, pp. 834–841.

[39] A. O. Abayomi, A. A. Olukayode, and G. O. Olakunle, "An overview of cache memory in memory management," *Automation, Control and Intelligent Systems*, vol. 8, no. 3, pp. 24–28, 2020. DOI: 10.11648/j.acis.20200803.11 eprint: https://article.sciencepublishinggroup.com/pdf/10.11648.j.acis.20200803.11. [Online]. Available: https://doi.org/10.11648/j.acis.20200803.11

[40] Y. Guo, X. Cao, L. Bainian, and M. Gao, "Cloud detection for satellite imagery using attention-based u-net convolutional neural network," *Symmetry*, vol. 12, p. 1056, Jun. 2020. DOI: 10.3390/sym12061056

[41] S.-H. Nam et al., "Nsct-based robust and perceptual watermarking for dibr 3d images," *IEEE Access*, vol. 8, pp. 93 760–93 781, 2020. DOI: 10.1109/ACCESS.2020.2994966

[42] R. Ranftl, K. Lasinger, D. Hafner, K. Schindler, and V. Koltun, "Towards robust monocular depth estimation: Mixing datasets for zero-shot cross-dataset transfer," *IEEE transactions on pattern analysis and machine intelligence*, vol. 44, no. 3, pp. 1623–1637, 2020.

[43] J. Taneja, Z. Liu, and J. Regehr, "Testing static analyses for precision and soundness," in *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*, ser. CGO '20, New York, NY, USA: Association for Computing Machinery, 2020, pp. 286–297, ISBN: 9781450370479. DOI: 10.1145/3368826.3377927

[44] Y. Zhao, L. Wang, S. Qi, W. Wang, and Q. Jia, "Quality evaluation of dibr-synthesized images based on holes and expanded regions," *IEEE Access*, vol. 8, pp. 10 640–10 648, 2020. DOI: 10.1109/ACCESS.2019.2963498

[45] J. Zhou et al., "Graph neural networks: A review of methods and applications," *AI open*, vol. 1, pp. 57–81, 2020.

[46] A. Aggarwal, M. Mittal, and G. Battineni, "Generative adversarial network: An overview of theory and applications," *International Journal of Information Management*, p. 100 004, Jan. 2021. DOI: 10.1016/j.jjimei.2020.100004

[47] X. Lyu et al., "Hr-depth: High resolution self-supervised monocular depth estimation," *Proceedings of the AAAI Conference on Artificial Intelligence*, 2021. [Online]. Available: https://cdn.aaai.org/ojs/16329/16329-13-19823-1-2-20210518.pdf

[48] Z. Wang, M. M. Swift, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Shf: A fast, accurate, and memory-efficient model for learning execution patterns from distributed traces," in *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, Virtual Event: USENIX Association, 2021, pp. 559–574, ISBN: 978-1-939133-23-6. [Online]. Available: https://www.usenix.org/conference/atc21/presentation/wang-zheng

[49] K. Han, Y. Wang, J. Guo, Y. Tang, and E. Wu, "Vision gnn: An image is worth graph of nodes," *Advances in neural information processing systems*, vol. 35, pp. 8291–8303, 2022.

[50] R. Szeliski, *Computer vision: algorithms and applications*. Springer Nature, 2022.

[51] S. Wei et al., *Unsupervised galaxy morphological visual representation with deep contrastive learning*, Nov. 2022. DOI: 10.48550/arXiv.2211.07168

[52] X. Shen et al., "Deepmad: (model table reporting resnet-18 parameters and flops)," 2023. [Online]. Available: https://arxiv.org/pdf/2303.02165

[53] S. Yang, Y. Zhou, X. Chen, C. Li, and H. Song, "Fault diagnosis for wind turbines with graph neural network model based on one-shot learning," *Royal Society Open Science*, vol. 10, Jul. 2023. DOI: 10.1098/rsos.230706

[54] K. Giannis, C. Thon, G. Yang, A. Kwade, and C. Schilde, "Predicting 3d particles shapes based on 2d images by using convolutional neural network," *Powder Technology*, vol. 432, p. 119 122, 2024, ISSN: 0032-5910. DOI: https://doi.org/10.1016/j.powtec.2023.119122 [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0032591023009051

[55] S. S. Mamand and A. I. Abdulla, "2d to 3d image conversion algorithms," in *ITM Web of Conferences*, EDP Sciences, vol. 64, 2024, p. 01 010.

[56] S. Singla, P. Saini, K. Malhotra, M. Kukreja, G. Kiran, and R. Bhandari, "Exploration and analysis of various techniques used in 2d to 3d image and video reconstruction," in *2024 International Conference on Emerging Innovations and Advanced Computing (INNOCOMP)*, 2024, pp. 8–15. DOI: 10.1109/INNOCOMP63224.2024.00013