Seeing Depth Through Frequency and Motion: A Progressive Training Paradigm for Monocular Depth Estimation

Supplementary Material

1. Theoretic Reserve

1.1. Proof of the Reciprocal Pose Constraint

In the section "Progressive Three-Stage Training Strategy" of the main paper, we proposed a reciprocal pose constraint for the PoseQuery Network to enforce geometric consistency between the forward and backward poses of adjacent frames. Specifically, the pose loss defined in Eq. (4) of the main paper is:

$$L_{\text{pose}} = \|R_{t\to s}R_{s\to t} - I\|_1 + \|R_{s\to t}t_{t\to s} + t_{s\to t}\|_2, \quad (1)$$

where R and t denote the rotation and translation components of the SE(3) transformations. This supplementary section provides the detailed derivation for the geometric constraints underlying this loss.

Lemma 1 (Inverse Composition of SE(3) Transformations)**.** *Given two frames A and B, let $T_{A\to B}$ and $T_{B\to A}$ be rigid-body transformations in $SE(3)$. Then:*

$$T_{B\to A}T_{A\to B} = I_4, \quad T_{A\to B}T_{B\to A} = I_4.$$

Proof. By definition of a rigid transformation, mapping any point $\mathbf{p}_A$ from frame A to frame B and back to A must leave the point unchanged:

$$\mathbf{p}_A = T_{B\to A}T_{A\to B}\mathbf{p}_A.$$

Since this holds for any $\mathbf{p}_A$, it follows that $T_{B\to A}T_{A\to B} = I_4$. Similarly, we can derive $T_{A\to B}T_{B\to A} = I_4$. □

Theorem 1 (Reciprocal Pose Constraints)**.** *Let $T_{A\to B}$ and $T_{B\to A}$ be expressed as:*

$$T = \begin{bmatrix} R & \mathbf{t} \\ \mathbf{0}^T & 1 \end{bmatrix}, \quad R \in SO(3), \mathbf{t} \in \mathbb{R}^3.$$

Then the following constraints must hold:

$$R_{A\to B}R_{B\to A} = I, \quad (2)$$

$$R_{B\to A}\mathbf{t}_{A\to B} + \mathbf{t}_{B\to A} = \mathbf{0}. \quad (3)$$

Proof. Expanding the product $T_{A\to B}T_{B\to A} = I_4$ gives:

$$\begin{bmatrix} R_{A\to B}R_{B\to A} & R_{A\to B}\mathbf{t}_{B\to A} + \mathbf{t}_{A\to B} \\ \mathbf{0}^T & 1 \end{bmatrix} = \begin{bmatrix} I_3 & \mathbf{0} \\ \mathbf{0}^T & 1 \end{bmatrix}.$$

From the top-left block:

$$R_{A\to B}R_{B\to A} = I_3.$$

From the top-right block:

$$R_{A\to B}\mathbf{t}_{B\to A} + \mathbf{t}_{A\to B} = \mathbf{0}.$$

Left-multiplying by $R_{B\to A}$ (the inverse of $R_{A\to B}$) yields:

$$R_{B\to A}\mathbf{t}_{A\to B} + \mathbf{t}_{B\to A} = \mathbf{0}.$$

□

Corollary 1 (Pose Loss Formulation)**.** *Minimizing the deviations from these constraints yields the pose loss:*

$$L_{pose} = \|R_{t\to s}R_{s\to t} - I\|_1 + \|R_{s\to t}t_{t\to s} + t_{s\to t}\|_2.$$

The L1-norm is used for the rotation term to improve robustness to outliers, while the L2-norm on translation encourages smooth regression of translational components.

1.2. Convergence View: Progressive Training as Non-convex Inexact BCD

Let the joint objective be $f(\theta_p, \theta_d)$. Our progressive strategy alternates between:

Pose update: optimize θ_p with θ_d fixed;

Depth update: optimize θ_d with θ_p fixed.

Because the objective f is a highly non-convex function parameterized by deep neural networks, computing the exact global minimum for each subproblem is computationally intractable. Instead, our progressive optimization alternates between the following two subproblems by performing *inexact* updates using gradient-based optimizers (e.g., Adam):

$$\begin{aligned} \theta_p^{k+1} &\approx \arg\min_{\theta_p} \ f(\theta_p, \theta_d^k), \\ \theta_d^{k+1} &\approx \arg\min_{\theta_d} \ f(\theta_p^{k+1}, \theta_d). \end{aligned} \quad (4)$$

In multivariable joint optimization problems with high-dimensional non-convex landscapes, directly performing full joint optimization often suffers from slow convergence or being trapped in poor local minima. The Block Coordinate Descent (BCD) method provides an efficient "divide-and-conquer" strategy. Specifically, our training paradigm aligns with the framework of Non-convex Inexact BCD [5]. Its core definition is as follows.

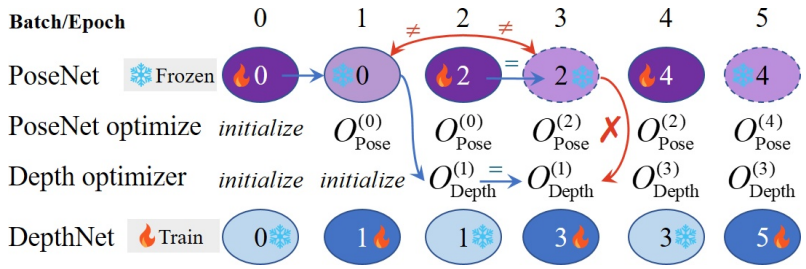

Figure S1. Illustration of optimizer-state misalignment in Batch/Epoch alternating training strategies. During alternating updates, the DepthNet or PoseNet is frequently frozen and unfrozen, but their corresponding optimizers retain outdated internal states (e.g., momentum, adaptive statistics), causing misalignment with the current network parameters.

Let the optimization problem be

$$\min_{x=(x_1,\dots,x_m)} f(x),$$

where $x \in \mathbb{R}^n$ is an n-dimensional optimization variable vector, partitioned into m non-overlapping *blocks*, i.e.,

$$\begin{aligned} &x = (x_1, x_2, \dots, x_m), \\ &\bigcup_{i=1}^{m} \text{supp}(x_i) = \{1, 2, \dots, n\}, \\ &\text{supp}(x_i) \cap \text{supp}(x_j) = \emptyset, \quad i \neq j, \end{aligned}$$

where $\text{supp}(x_i)$ denotes the index set of the i-th block of variables.

Unlike classical exact BCD, the convergence theory for inexact and non-convex BCD does not require finding the exact global minimum at each step. Instead, it relies on the following standard assumptions [1, 5]:

Assumption 1 (Sufficient Descent): Rather than requiring exact minimization, the inexact iterative sequence $\{x^k\}$ ensures a sufficient decrease in the objective function:

$$f(x^{k+1}) \leq f(x^k) - \alpha \|x^{k+1} - x^k\|^2, \quad \alpha > 0. \tag{5}$$

Assumption 2 (Gradient-related Updates): The updates at each iteration are bounded by the gradients:

$$\|\nabla f(x^{k+1})\| \leq c \|x^{k+1} - x^k\|, \quad c > 0. \tag{6}$$

Assumption 3 (Bounded Level Sets and Lipschitz Gradients): The level set $\{x : f(x) \leq f(x^0)\}$ is bounded, and the block-wise partial gradients of f are Lipschitz continuous.

These assumptions are routinely satisfied in our setting: executing a finite number of optimization steps using Adam with stable learning rates ensures the sufficient descent (Assumption 1) and gradient-related updates (Assumption 2). Furthermore, deep neural network objectives constructed with standard piecewise linear activations (e.g., ReLU) are semi-algebraic, meaning they naturally satisfy the Kurdyka-Lojasiewicz (KL) property.

Under these conditions, classical results from non-convex inexact block coordinate descent theory guarantee that the iterative sequence $\{x^k\}$ satisfies:

1. **Monotonic decrease of the objective:**

$$f(x^{k+1}) \leq f(x^k), \quad \forall k \geq 0. \tag{7}$$

2. **Convergence to a stationary point:** Any limit point x^* of the sequence $\{x^k\}$ is a critical/stationary point of f, i.e.,

$$\nabla f(x^*) = 0 \quad \text{or equivalently} \quad 0 \in \partial f(x^*). \tag{8}$$

3. **Global sequence convergence under KL property:** Because f satisfies the KL property, the entire sequence $\{x^k\}$ is guaranteed to converge to a stationary point x^*:

$$\lim_{k \to \infty} x^k = x^*. \tag{9}$$

Therefore, by strictly satisfying the conditions of Non-convex Inexact BCD, our progressive training strategy inherits its theoretical convergence guarantees. This rigorously explains its improved stability and effectiveness compared to single-stage joint optimization.

2. Further Analysis

2.1. Analysis of Alternating Training Instability

In the section "Effectiveness of the Progressive Training Strategy" of the main paper, we briefly mentioned that the Batch and Epoch alternating strategies suffer from misaligned learning dynamics, which hinder model convergence. Here, we provide a detailed explanation supported by an illustrative diagram (Fig. S1).

Optimizer-State Misalignment. Consider the Epoch alternating strategy: at each epoch, one network (DepthNet or PoseNet) is frozen while the other is trained. For instance, at the beginning of Epoch 3 (Fig. S1), the PoseNet has just been updated to $\text{Pose}^{(2)}$ in Epoch 2, and now the DepthNet is unfrozen for training. However, the Depth optimizer $O^{(1)}_{\text{depth}}$ still retains its internal state (e.g., momentum,

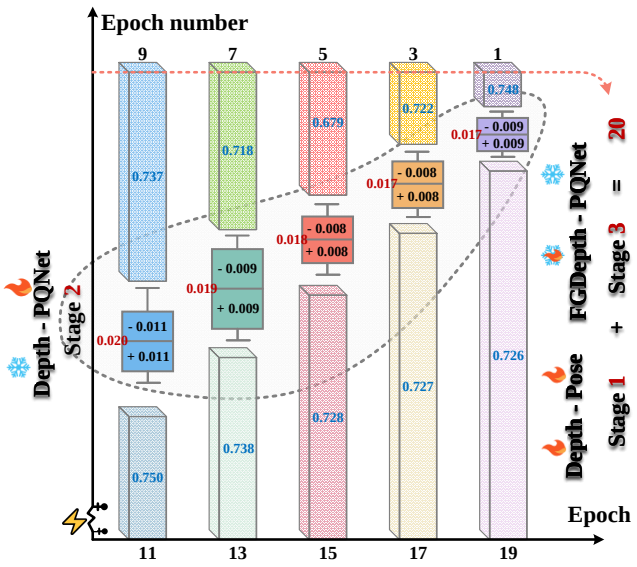

Figure S2. Ablation on epoch allocation in the proposed three-stage progressive training strategy

adaptive second-order statistics) from Epoch 1, which was computed when the DepthNet was coupled with an outdated frozen PoseNet $\text{Pose}^{(0)}$.

This leads to a critical mismatch between the optimizer state and the current network state, since the Depth optimizer's accumulated gradients and moments are no longer compatible with the updated PoseNet. As a result, the optimizer's updates push the model parameters along a direction that does not align with the new coupled task dynamics, which not only fails to improve performance but may even degrade it. We refer to this phenomenon as optimizer-state misalignment.

Impact on Convergence. This misalignment explains why the Batch and Epoch alternating strategies, which frequently freeze and unfreeze networks, lead to unstable training and poor convergence. In contrast, our Stage-wise progressive training fully resets and realigns the optimization states at each stage, avoiding these mismatches and achieving significantly better convergence and performance (Table S1).

This analysis highlights the importance of aligning optimizer states with network updates, further motivating the design of our stage-wise progressive training paradigm.

2.2. Detailed Analysis of the CAA Mechanism

While CAA is conceptually inspired by attention, it is structurally distinct from standard Cross-Attention (SCA) due to its strict channel-wise alignment and local motion constraints. Prior attention models utilizing SCA compute similarity by projecting features across the entire channel dimension. This dense, all-to-all channel mixing introduces significant computational overhead and potential feature redundancy. In contrast, CAA restricts interactions to strictly

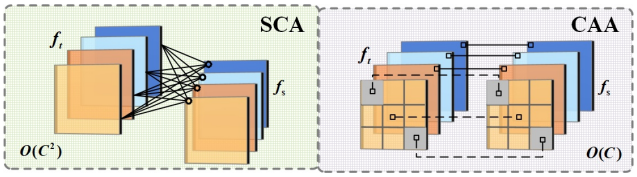

Figure S3. SCA vs. CAA (Ours).

aligned channels. This structural constraint prevents the entanglement of unrelated channel semantics, ensuring that specific motion dynamics precisely guide their corresponding depth features. The complexities are compared as follows:

- **SCA Complexity:** $\mathcal{O}(C^2)$ (dense channel mixing).
- **CAA Complexity:** $\mathcal{O}(C)$ (channel-aligned interaction).

This reduction to linear complexity $\mathcal{O}(C)$ along the channel dimension ensures highly efficient feature alignment while preserving high-frequency structural details, as visually depicted in Fig. S3. Furthermore, by replacing dense channel mixing with aligned interactions and adopting local 7×7 windows, CAA significantly reduces the overall computational burden.

3. Ablation and Complexity Analysis of the Progressive Training Strategy

3.1. Ablation on Epoch Allocation

To analyze how each stage of our progressive training strategy contributes to final depth performance, we conduct a detailed ablation on the allocation of training epochs. We fix the combined budget of Stage 1 and Stage 3 to 20 epochs (the same as the standard joint-training schedule) and vary their split across several configurations (11–9, 13–7, 15–5, 17–3, 19–1). Stage 2 is shortened correspondingly to maintain a comparable overall training cost.

In Fig. S2, Stage 1 and Stage 3 curves report the Sq Rel metric, while the Stage 2 curve reflects the average trajectory error of the pose network. As shown in the figure, different allocations lead to noticeable variations in performance. Insufficient Stage 1 training weakens the initialization of both depth and pose networks, whereas allocating too few epochs to Stage 3 reduces the decoder's capacity to refine high-frequency structures. The best performance is achieved near the 15/5 split, which balances stable optimization with effective structural refinement. This analysis supports the final 15/10/5 configuration adopted in our full model.

3.2. Computational Complexity

Our progressive training scheme replaces the standard 20-epoch joint optimization with three sequential stages of 15/10/5 epochs. Although this strategy yields significant performance gains, it slightly increases training cost due to forward-only passes through the frozen network. This section provides a precise analysis of the resulting computa-

Table S1. Ablation study on different training strategies for our combined FGDepth + PoseQuery Network model.

Training Strategy	Lower is better↓				Higher is better↑		
	Abs Rel	Sq Rel	RMSE	RMSE log	$\delta < 1.25^1$	$\delta < 1.25^2$	$\delta < 1.25^3$
Joint	0.098	0.704	4.338	**0.174**	0.901	**0.967**	**0.984**
Batch	0.117	0.819	4.448	0.185	0.875	0.957	0.983
Epoch	0.098	0.710	4.381	0.175	0.901	0.966	**0.984**
Stage	**0.096**	**0.679**	**4.303**	**0.174**	**0.902**	**0.967**	**0.984**

tional complexity. Importantly, this overhead affects training only; inference-time computation, FLOPs, and runtime remain unchanged.

We denote by $O(n_{\text{depth}})$ the total computational cost of training the depth network for 20 epochs under the standard joint-training scheme, and by $O(n_{\text{pose}})$ the cost for the pose network.

A full training step (forward + gradient computation + backward) is approximated as three forward passes. When a network is frozen, it participates only through a single forward pass, i.e., $1/3$ of its normal cost.

The progressive schedule is:

Stage 1: 15 epochs, train both networks.

Stage 2: 10 epochs, freeze depth network, train PQNet.

Stage 3: 5 epochs, freeze PQNet, fine-tuning FGDepth.

Depth Network Complexity. The total cost for the depth network is:

$$\begin{aligned} O(n_{\text{FGDepth}}) &= \frac{15}{20}O(n_{\text{depth}}) + \frac{1}{3}\cdot\frac{10}{20}O(n_{\text{depth}}) + \frac{5}{20}O(n_{\text{depth}}), \\ &= \left(1+\frac{1}{6}\right)O(n_{\text{depth}}). \end{aligned} \tag{10}$$

The three-stage training of the depth network requires a total computational cost of approximately $7/6$ of standard joint training, corresponding to only a $1/6$ increase.

Pose Network Complexity. Similarly, the pose network cost is:

$$\begin{aligned} O(n_{\text{PQNet}}) &= \frac{15}{20}O(n_{\text{pose}}) + \frac{10}{20}O(n_{\text{pose}}) + \frac{1}{3}\cdot\frac{5}{20}O(n_{\text{pose}}), \\ &= \left(\frac{5}{4}+\frac{1}{12}\right)O(n_{\text{pose}}) = \frac{4}{3}O(n_{\text{pose}}). \end{aligned} \tag{11}$$

For the pose network, the three-stage schedule amounts to roughly $4/3$ of the computation used for the PoseNet updates in the 20-epoch joint-training baseline, i.e., an increase of $1/3$. However, since PoseNet is a lightweight ResNet18-based module, its contribution to the overall training cost remains relatively small.

In addition, Table S3 reports FLOPs, parameter counts, and inference latency for all model variants. These results show that introducing the plug-and-play FGS block has only a minor impact on computational complexity, while consistently improving depth accuracy across backbones. This highlights the efficiency and practicality of FGS as a lightweight enhancement module rather than a computational burden.

4. Dataset Description

For clarity, we summarize the datasets used in our experiments (details are provided in Section "Datasets" of the main paper):

- **KITTI Depth [2]:** Eigen split (39,810 training, 4,424 validation, 697 test images), using raw LiDAR as ground truth.
- **Make3D [4]:** 134 images for zero-shot generalization testing with provided ground truth depth.
- **KITTI Odometry [2]:** Official odometry split with 11 sequences; sequences 09–10 used for evaluation only.
- **DrivingStereo [7]:** A large-scale stereo dataset designed for autonomous driving, containing over 170,000 image pairs with dense depth maps generated from stereo matching. It includes four subsets captured under different weather conditions (foggy, cloudy, rainy, sunny), each containing 500 images. Following prior works, all models are trained on KITTI and evaluated on DrivingStereo without any fine-tuning to assess cross-dataset generalization.

5. Additional Experimental Results

5.1. Results on KITTI

We provide additional qualitative results on the KITTI dataset in Fig. S4. These results complement those presented in the "Comparison with State-of-the-Art" section of the main paper and further showcase our method's strong capability to capture fine-grained structures in complex driving scenes, effectively modeling subtle depth variations and delineating object boundaries. In particular, as illustrated in the top four rows of Fig. S4, our approach consistently outperforms competing methods on small or thin objects, further confirming its effectiveness in preserving delicate structural details.

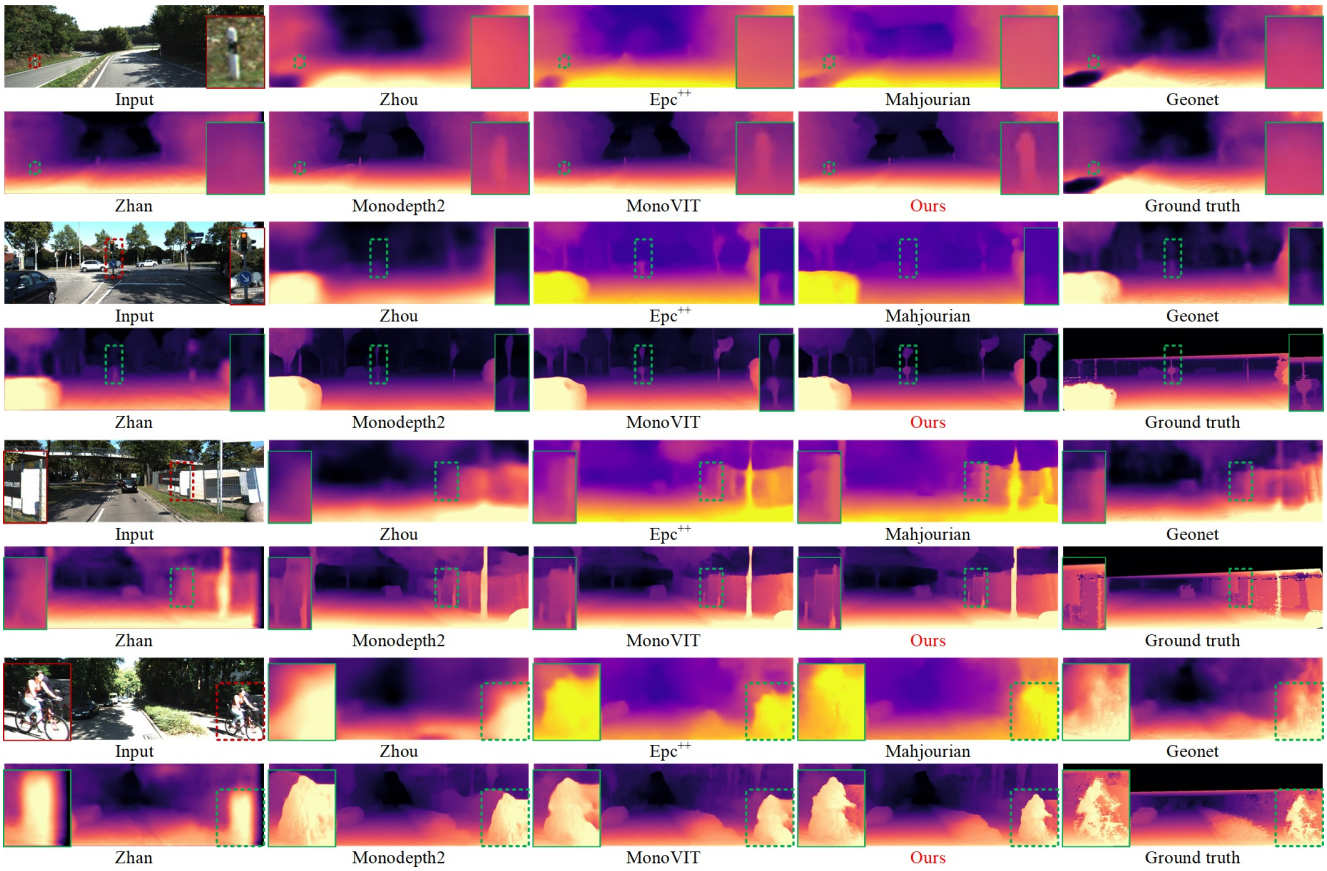

Figure S4. Additional qualitative depth estimation results on the KITTI dataset. Our method produces sharper object boundaries and more coherent depth predictions compared to seven representative baselines.

5.2. High-Resolution KITTI Experiments

To further evaluate the scalability of our frequency-guided design, we assess a high-resolution variant of our model, denoted as FGDepth*, at 1024×320 resolution. FGDepth* incorporates only the proposed FGS blocks into the baseline depth network, while excluding PQNet and the progressive training strategy. This setting isolates the contribution of FGS blocks when provided with higher-resolution inputs.

As shown in Table S2, FGDepth* achieves consistent improvements over previous self-supervised methods across all metrics. The gains are especially pronounced in Sq Rel and δ-accuracy, indicating that higher-resolution imagery provides richer high-frequency details and that FGS blocks effectively preserve such structures by mitigating aliasing. These results demonstrate that the benefits of our FGS design extend naturally to high-resolution settings.

5.3. Zero-Shot Generalization on Make3D

In addition, Fig. S5 presents zero-shot predictions on the Make3D dataset [4]. These results highlight our model's robustness to unseen environments and its satisfactory generalization performance when transferring across domains with significant scene and appearance differences.

5.4. Visualization of Pose Trajectories

To complement the quantitative results reported in the section "Effectiveness of FGDepth and PQNet Models" of the main paper, we provide additional qualitative visualizations of pose trajectories for different variants of our method on the KITTI odometry sequences 09 and 10. Figure S6 presents additional trajectory visualizations on the KITTI odometry sequences 09 and 10 for our two variants (PQNet and PQNet+Stage), along with Monodepth2 as a baseline. The corresponding quantitative results are reported in Table 5 of the main paper, where PQNet+Stage achieves the best overall performance. Consistently, the visualized trajectories also confirm this observation: while PQNet performs comparably to Monodepth2 on sequence 09, it shows a clear improvement on sequence 10. Moreover, PQNet+Stage yields the most accurate and stable trajectories on both sequences, further validating its effectiveness.

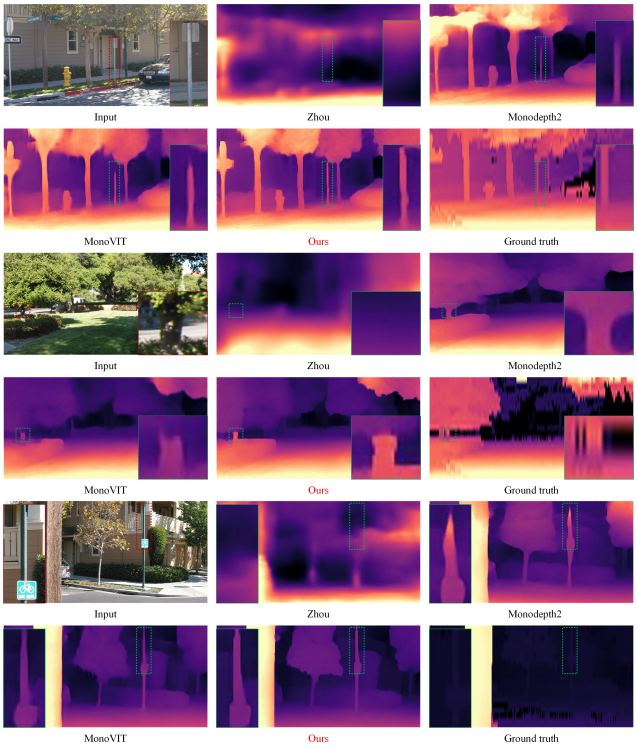

Figure S5. Zero-shot generalization results on Make3D. Our model trained only on KITTI adapts well to the new domain without fine-tuning.

Table S2. Quantitative results on KITTI (1024×320 training and testing) and zero-shot generalization results on DrivingStereo [7] using KITTI-trained 640×192 models. FGDepth delivers superior accuracy at high resolution and stronger robustness under diverse weather conditions.

Method	Lower is better↓				Higher is better↑		
	Abs Rel	Sq Rel	RMSE	RMSE log	δ_1	δ_2	δ_3
KITTI: 1024 × 320 resolution.							
Monodepth2 [3]	0.115	0.882	4.701	0.190	0.879	0.961	0.982
R-MSFM6 [10]	0.108	0.748	4.470	0.185	0.889	0.963	0.982
CAdepth [6]	0.102	0.734	4.407	0.178	0.898	0.966	0.984
Lite-Mono [8]	0.102	0.746	4.444	0.179	0.896	0.965	0.983
MonoViT [9]	**0.096**	0.714	4.292	**0.172**	**0.908**	0.968	0.984
FGDepth* (ours)	**0.096**	**0.684**	**4.252**	**0.172**	**0.908**	**0.969**	**0.985**
DrivingStereo Dataset:foggy, 640×192							
Monodepth2 [3]	0.142	1.940	9.823	0.218	0.813	0.937	0.974
Lite-Mono [8]	0.135	1.741	9.383	0.207	0.820	0.944	0.979
MonoViT [9]	0.110	1.222	7.801	0.168	0.869	0.966	0.990
FGDepth (Ours)	**0.104**	**1.089**	**7.372**	**0.160**	**0.878**	**0.970**	**0.991**
DrivingStereo Dataset: cloudy, 640×192							
Monodepth2 [3]	0.168	2.169	8.419	0.231	0.783	0.933	0.974
Lite-Mono [8]	0.148	1.625	7.728	0.214	0.806	0.944	0.979
MonoViT [9]	0.140	1.606	7.529	0.201	0.831	0.948	0.981
FGDepth (Ours)	**0.136**	**1.444**	**7.287**	**0.196**	**0.831**	**0.952**	**0.984**
DrivingStereo Dataset: rainy, 640×192							
Monodepth2 [3]	0.240	3.533	12.214	0.306	0.607	0.857	0.948
Lite-Mono [8]	0.200	2.729	10.981	0.265	0.686	0.890	0.966
MonoViT [9]	0.173	2.106	9.610	0.230	0.736	0.934	0.980
FGDepth (Ours)	**0.172**	**2.040**	**9.348**	**0.227**	**0.738**	**0.931**	**0.982**
DrivingStereo Dataset: sunny, 640×192							
Monodepth2 [3]	0.177	2.073	8.165	0.241	0.783	0.924	0.967
Lite-Mono [8]	0.160	1.674	7.598	0.224	0.803	0.936	0.973
MonoViT [9]	0.149	1.564	7.564	0.211	0.818	0.943	0.978
FGDepth (Ours)	**0.140**	**1.382**	**7.267**	**0.204**	**0.826**	**0.949**	**0.980**

5.5. Generalization to DrivingStereo under Diverse Weather Conditions

To further assess the robustness and generalization capability of our model, we evaluate FGDepth on the DrivingStereo dataset [7], which includes four subsets captured under foggy, cloudy, rainy, and sunny conditions. Following prior works such as MonoViT, all models are trained solely on the KITTI dataset and directly tested on DrivingStereo without any fine-tuning.

As shown in Table S2, our method achieves consistent improvements over recent self-supervised approaches across all four weather conditions. The most notable gains appear in the foggy subset, where visibility is severely reduced: FGDepth lowers Abs Rel from 0.110 to 0.104 and RMSE from 7.801 to 7.372, demonstrating stronger robustness under low-visibility conditions. Similar improvements are observed in rainy and sunny scenes, indicating that our approach preserves structural cues more effectively and reduces failure cases under domain shift.

Since prior works do not provide an official evaluation pipeline for DrivingStereo, we follow publicly available instructions in the MonoViT [9] repository and implement a custom data loader while reusing the Monodepth2 [3] ground-truth processing and evaluation protocol. This ensures a consistent and fair comparison across all publicly released models.

6. Extended Applications of the FGS Block

To further demonstrate the plug-and-play and architecture-agnostic nature of the proposed FGS block, we integrate it into three representative monocular depth estimation frameworks: (1) the classic Monodepth2 with a ResNet18 encoder, (2) the lightweight Lite-Mono network, and (3) the MPViT-small transformer backbone used in our main experiments.

6.1. Integration into Diverse Backbone Networks

For MPViT-small, we replace its original decoder with the FGS-enhanced version using the native channel configuration. For ResNet18 and Lite-Mono, we simply adapt the decoder's channel layout to match each backbone's feature hierarchy, without modifying any part of the encoder. This enables a clean, drop-in replacement requiring no architectural redesign.

As shown in Table S3, introducing FGS blocks consistently improves all depth evaluation metrics across all backbones. The most notable gains occur in Sq Rel, which reflects large-structure depth accuracy: FGS reduces Sq Rel

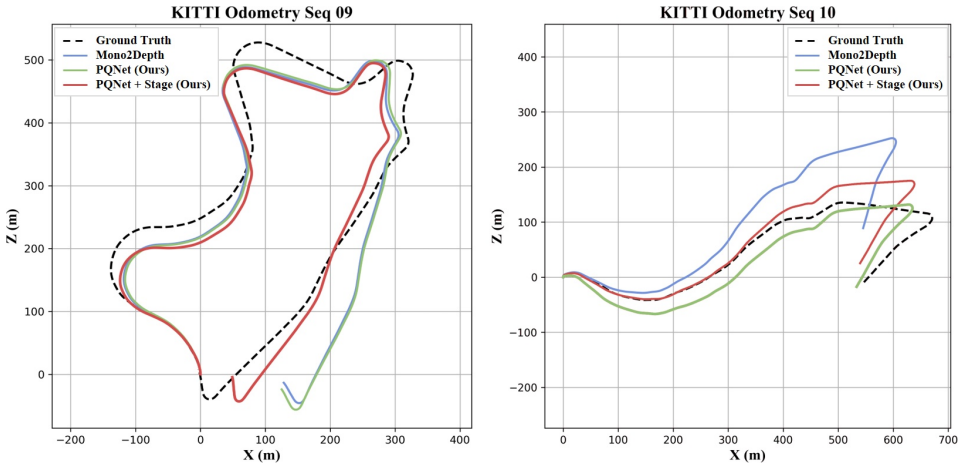

Figure S6. Trajectory visualization for KITTI odometry sequences 09 and 10.

Table S3. Integration of the proposed FGS block into three different backbone networks on the KITTI Eigen split (640 × 192). All models are trained with monocular supervision. Introducing FGS consistently improves performance across all metrics and backbones, demonstrating the plug-and-play nature and strong generality of our design.

Method	Abs Rel ↓	Sq Rel↓	RMSE↓	RMSE log ↓	δ_1↑	δ_2 ↑	δ_3 ↑	**Parameters**	**FLOPs**	**Training**	**Inference**
ResNet18	0.115	0.903	4.863	0.193	0.877	0.959	0.981	14.84 M	8.04 G	5h06m	4.21 ms
ResNet18+FGS Block	**0.108**	**0.784**	**4.633**	**0.185**	**0.883**	**0.962**	**0.982**	14.98 M	9.25 G	6h19m	8.35 ms
Lite-mono	0.107	0.808	4.622	0.187	0.879	0.961	0.982	3.07 M	5.03 G	7h07m	8.45ms
Lite-mono+FGS Block	**0.103**	**0.727**	**4.452**	**0.183**	**0.886**	**0.964**	**0.983**	3.13 M	5.31 G	9h26m	10.38ms
Mpvit-small	0.102	0.725	4.428	0.176	0.895	**0.966**	0.982	27.87 M	-	14h38m	39.36 ms
Mpvit-small+FGS Block	**0.097**	**0.687**	**4.323**	**0.175**	**0.902**	**0.966**	**0.984**	28.14 M	-	16h18m	49.23ms

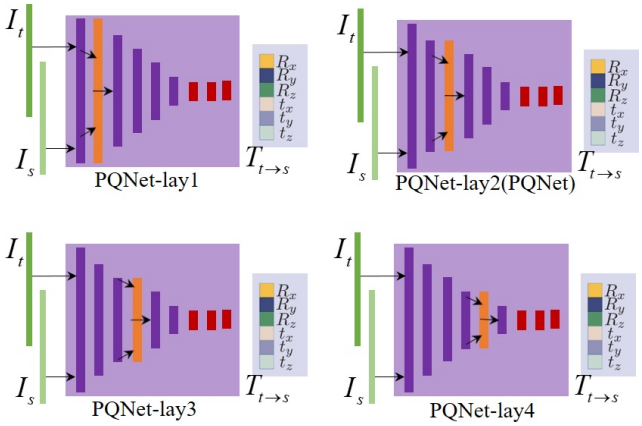

Figure S7. Effect of inserting the PQLayer at different stages of the ResNet18 backbone for pose estimation. Subfigures correspond to inserting the PQLayer after layers 1–4. Early insertion (layer 1) fails to provide meaningful motion cues, while late insertion (layer 4) leads to unstable optimization. Insertion after layer 2 achieves the best balance between positional information retention and training stability.

by 13.2% on ResNet18, 10.0% on Lite-Mono, and 5.2% on MPViT-small. These substantial improvements—despite the architectural diversity among the backbones—indicate that our dual-path frequency modeling generalizes robustly across both CNN- and Transformer-based frameworks.

6.2. Computational Efficiency Analysis

In addition to accuracy improvements, the FGS block maintains strong computational efficiency. Although frequency-guided operations introduce slight additional overhead, the resulting latency increase remains small (e.g., approximately 4.14 ms for ResNet18 and 1.93 ms for Lite-Mono). Similarly, parameter count and FLOPs increase only marginally across all backbones.

This favorable cost–performance trade-off highlights the practicality of the FGS block for real-world deployment, especially in latency-sensitive settings such as robotics or autonomous driving. Combined with the high-resolution KITTI experiments in Sec. 5.2, these results demonstrate that FGS generalizes effectively across architectures and input resolutions, reinforcing its versatility as a compact and powerful drop-in module for monocular depth estimation.

Table S4. Ablation study on the placement of the PQLayer in two backbones: ResNet18 and Swin-Transformer. Results are average trajectory errors on KITTI odometry sequence 09 (lower is better).

ResNet18 (Monodepth2)	
Baseline (Without PQLayer)	0.021 ± 0.011
+ PQLayer after layer 1	0.021 ± 0.010
+ PQLayer after layer 2	$0.018 \pm \mathbf{0.008}$
+ PQLayer after layer 3	$\mathbf{0.017} \pm 0.009$
+ PQLayer after layer 4	0.024 ± 0.012
Swin-Transformer (672×672)	
Baseline (Without PQLayer)	$0.020 \pm \mathbf{0.009}$
+ PQLayer after layer 2	$\mathbf{0.018} \pm 0.012$

7. Extended Applications of PQLayer

7.1. Application to Different Layers of the Current Network

We conduct an in-depth analysis of integrating the Pose Query Layer (PQLayer) at different stages of a ResNet18 backbone to investigate its impact on performance. The structural configurations for different insertion points are illustrated in Fig. S7. As shown in Table S4, inserting the PQLayer too early (e.g., after the first layer) results in performance comparable to a plain ResNet18, indicating that overly early fusion fails to provide meaningful motion cues. Conversely, inserting the PQLayer too late (e.g., after the fourth layer) also degrades performance, as features at this stage are compressed into low-resolution, high-level semantics, which cannot effectively capture motion-related information. Moreover, late-stage fusion introduces large inter-epoch variations during training, indicating unstable optimization.

In contrast, placing the PQLayer after the second or third layer produces more promising results, with similar final performance. However, from the perspective of training stability and convergence, we ultimately select inserting the PQLayer after the second layer as our default configuration for PQNet, balancing the retention of positional information and optimization stability. In the PQLayer, the query features are extracted from the previous frame, while the key and value features are taken from the current frame. This design allows the model to explicitly establish temporal correspondences between consecutive frames by querying semantically similar regions in the current frame's feature map based on information from the preceding frame. Such a cross-frame querying mechanism effectively leverages motion cues and enhances the network's ability to capture geometric and temporal consistency, which is particularly beneficial for pose estimation tasks.

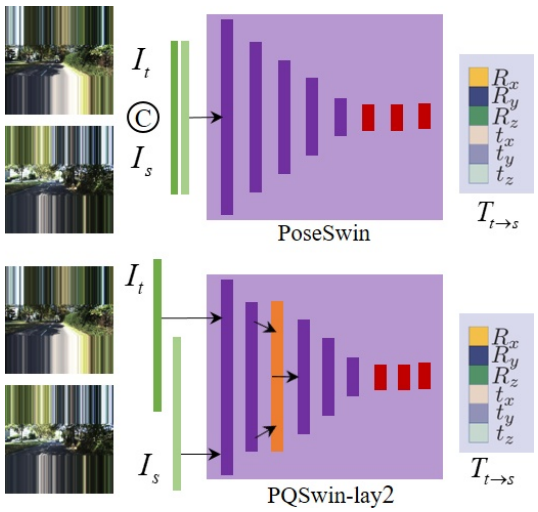

Figure S8. Integration of the PQLayer into the Swin-Transformer backbone for pose estimation. The top subfigure shows the vanilla Swin-Transformer as a pose encoder, while the bottom subfigure illustrates the configuration with the PQLayer inserted after the second layer. All inputs are resized to 672×672 to meet the architectural requirements of the Swin-Transformer.

7.2. Application to Other Backbone Networks

We further evaluate the generality of the PQLayer by integrating it into a Swin-Transformer backbone. To meet the architectural requirements of the Swin-Transformer, input images are resized to a square resolution of 672×672, and the input channels are modified to six by concatenating the previous frame with the current frame to provide richer motion cues. Following the same strategy as with ResNet18, we insert the PQLayer after the second layer of the Swin-Transformer, enabling cross-frame querying between consecutive frames, as illustrated in Fig. S8.

Interestingly, the ResNet18 backbone with the PQLayer inserted after layer 2 achieves comparable or even slightly better performance than the Swin-Transformer configuration (Table S4), despite the latter incurring substantially higher computational costs. Considering this, we ultimately adopt ResNet18 as the backbone for PoseNet in our main experiments, striking a better balance between performance and efficiency. This may be attributed to the limited training data, which prevents the Swin-Transformer from fully utilizing its modeling capacity.

8. Hyperparameters and Tuning

Table S5 lists the full set of hyperparameters used in our experiments, along with their search ranges (if tuning was performed) and the final selected values. We follow standard practices in self-supervised monocular depth estimation, keeping most hyperparameters aligned with Monodepth2 [3] for fair comparison. Hyperparameters were

Table S5. Key hyperparameters used in our experiments.

Parameter	**Final Value**
Optimizer	AdamW ($\beta_1 = 0.9, \beta_2 = 0.999$, weight decay = 0)
Learning rate	$1e^{-4}$ (depth encoder: $5e^{-5}$)
LR scheduler	ExponentialLR (decay factor $\gamma = 0.9$)
Batch size	12
Epochs	Joint: 20; Progressive: 15+10+5 (three stages)
Loss weights	photometric: 1.0, smoothness: 0.001, pose: 1.0 (Stage 2 only)
Image resize	640×192 (KITTI)
Data augmentation	random flip, color jitter, resize

Table S6. Performance over 5 runs (mean ± std) of our method on KITTI (Eigen split).

Metric	Abs Rel	Sq Rel	RMSE	RMSE log
Ours (5 runs)	0.097 ± 0.001	0.682 ± 0.015	4.318 ± 0.025	0.175 ± 0.001

Table S7. Key software dependencies in our experimental environment.

Package	**Version**
python	3.8
pytorch	1.9.0
cudatoolkit	11.1
torchvision	0.10.0
numpy	1.21.6
opencv	4.6.0.66
matplotlib	3.5.3

selected based on validation performance on the KITTI dataset.

9. Experimental Environment

All experiments were conducted on a workstation with the following hardware and software configuration:

- **CPU:** Intel Core i9-10920X (12 cores, 24 threads)
- **GPU:** 2 × NVIDIA RTX 3090 (24GB VRAM each)
- **RAM:** 96GB
- **Operating System:** Ubuntu 20.04 LTS
- **Python:** 3.8 (Conda environment)
- **CUDA Toolkit:** 11.1
- **cuDNN:** 8.0.5 (bundled with PyTorch 1.9.0+cu111
- **Deep Learning Framework:** PyTorch 1.9.0

Table S7 summarizes the key software dependencies used in our experiments.

All packages were installed in a Conda environment; for complete reproducibility, the full list of dependencies is provided in the "fgdepth_env.yml" file included in the submitted code package.

10. Reproducibility and Training Stability

To assess the stability of our method, we trained the final model five times with different random seeds and report the mean and standard deviation of depth metrics on the KITTI Eigen split. Results in Table S6 indicate that our model achieves consistent performance with low variance across runs.

11. Code Availability and Randomness Settings

To ensure reproducibility, we provide the complete training and evaluation code for the single-stage joint training setup in the supplementary material (see the "code" folder). Upon acceptance, the full codebase will be publicly released. The released repository will include:

- Source code for all proposed modules (FGS Block, PQLayer, progressive training pipeline).
- Configuration files and scripts to reproduce all experiments reported in the paper.
- Pretrained model weights for the main experiments.
- Detailed instructions for dataset preparation, training, and evaluation.

Randomness settings. We use the same stochastic data augmentation techniques as Monodepth2 [3], implemented in our data loader:

- **Color augmentation:** Random brightness, contrast, saturation, and hue changes applied using "ColorJitter" with a 50% probability.
- **Horizontal flipping:** Random left-right flipping with a 50% probability.

Additionally, network weights are initialized using PyTorch's default initializers. We did not explicitly fix random seeds in our experiments; thus, minor variations in reported metrics may occur due to these stochastic augmentations and parameter initialization. For full repro-

ducibility, the final code release will provide options to set fixed random seeds and enable deterministic computation.

References

[1] Hedy Attouch, Jérôme Bolte, and Benar Fux Svaiter. Convergence of descent methods for semi-algebraic and tame problems: proximal algorithms, forward–backward splitting, and regularized gauss–seidel methods. *Mathematical Programming*, 137(1):91–129, 2013.

[2] Andreas Geiger, Philip Lenz, Christoph Stiller, and Raquel Urtasun. Vision meets robotics: The KITTI dataset. *International Journal of Robotics Research*, 32(11):1231–1237, 2013.

[3] Clément Godard, Oisin Mac Aodha, Michael Firman, and Gabriel J Brostow. Digging into self-supervised monocular depth estimation. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 3827–3837, 2019.

[4] Ashutosh Saxena, Min Sun, and Andrew Y. Ng. Make3D: Learning 3D scene structure from a single still image. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 31(5):824–840, 2009.

[5] Yangyang Xu and Wotao Yin. A block coordinate descent method for regularized multiconvex optimization with applications to nonnegative tensor factorization and completion. *SIAM Journal on Imaging Sciences*, 6(3):1758–1789, 2013.

[6] Jiaxing Yan, Hong Zhao, Penghui Bu, and YuSheng Jin. Channel-wise attention-based network for self-supervised monocular depth estimation. In *Proceedings of the International Conference on 3D Vision*, pages 464–473, 2021.

[7] Guorun Yang, Xiao Song, Chaoqin Huang, Zhidong Deng, Jianping Shi, and Bolei Zhou. Drivingstereo: A large-scale dataset for stereo matching in autonomous driving scenarios. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 899–908, 2019.

[8] Ning Zhang, Francesco Nex, George Vosselman, and Norman Kerle. Lite-Mono: A lightweight CNN and transformer architecture for self-supervised monocular depth estimation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 18537–18546, 2023.

[9] Chaoqiang Zhao, Youmin Zhang, Matteo Poggi, Fabio Tosi, Xianda Guo, Zheng Zhu, Guan Huang, Yang Tang, and Stefano Mattoccia. MonoViT: Self-supervised monocular depth estimation with a vision transformer. In *Proceedings of the International Conference on 3D Vision*, pages 668–678, 2022.

[10] Zhongkai Zhou, Xinnan Fan, Pengfei Shi, and Yuanxue Xin. R-MSFM: Recurrent multi-scale feature modulation for monocular depth estimating. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 12757–12766, 2021.