FalconApp: A Mobile Client–Server Pipeline for Auto-Labeled Object Perception

Anonymous Author(s)
Affiliation withheld for double-blind review

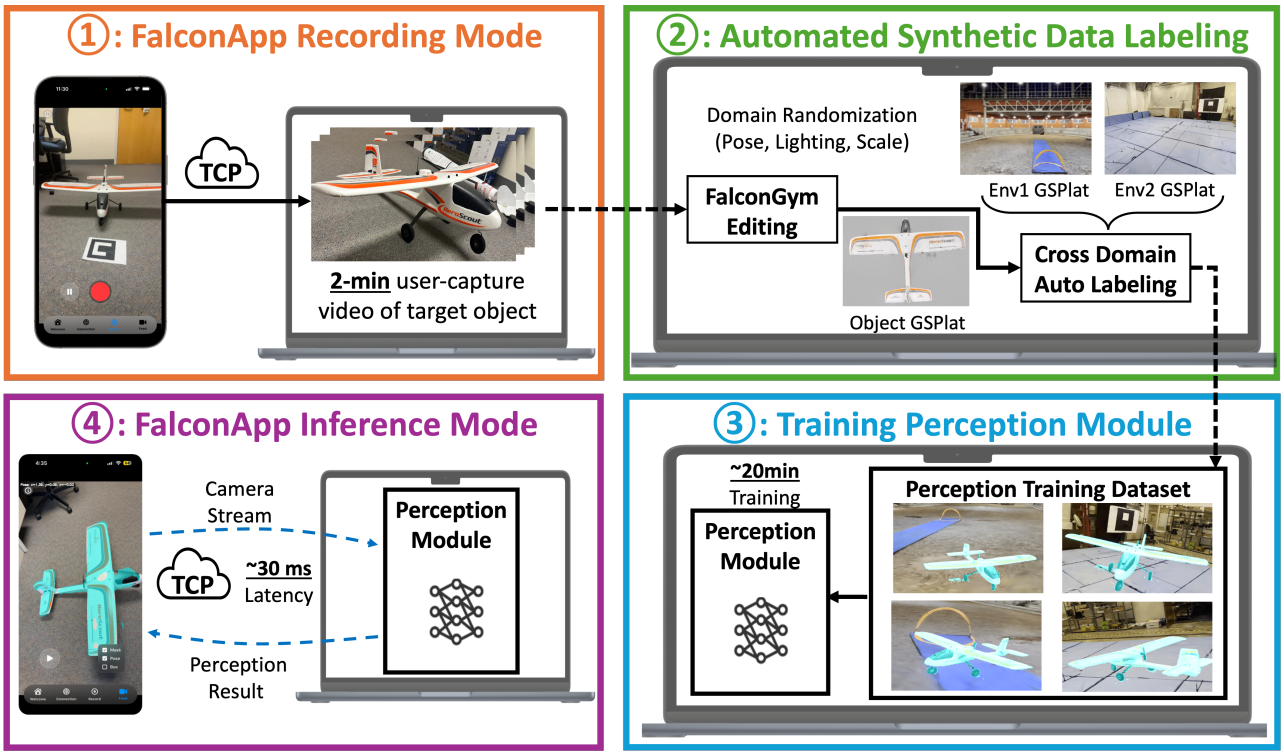

Fig. 1: **FalconApp Pipeline:** From a two-minute iPhone video, FalconApp reconstructs an object GSplat and composites it with diverse backgrounds in FalconGym 2.0 [1] to generate images with render-aligned masks and 6-DoF poses. Post-reconstruction data generation and training take 12–23 minutes. For live use, the app streams images to the backend and displays the returned predictions, with a measured client–server round trip of 29–32 ms.

***Abstract*— Onboarding a new rigid object into a learned perception system requires both labeled training data and the infrastructure that connects capture, training, and inference. We present FalconApp, a mobile client–server pipeline for this workflow. From a two-minute handheld phone capture, a GPU backend reconstructs an editable 3D Gaussian Splatting (GSplat) asset, composites it with diverse backgrounds, generates images with render-aligned masks and 6-DoF poses, and trains an object-specific perception module without per-image manual mask or pose labels. During live use, the phone streams images to the backend for offboard inference and displays the returned predictions. FalconApp reuses the rendering, auto-labeling, and perception components of FalconGym 2.0 and FalconTrack; the contribution of this work is the Swift mobile client, frontend–backend protocol, workflow orchestration, and system-level characterization of the integrated pipeline. Across five rigid object instances spanning diverse geometry, scale, and symmetry, post-reconstruction data generation and training take 12–23 minutes per object, and the measured client–server inference round trip is 29–32 ms. FalconApp has lower translation and angular errors than a geometric PnP baseline on four of five objects in simulation and the real world, and lower errors than a per-object-and-background-trained neural pose regressor on all three shared objects. For the rotationally symmetric lamp, simulated results are mixed and both real-world errors are worse than PnP, exposing a limitation of the current pose pipeline.**

I. Introduction

Onboarding a new rigid object into a learned perception system involves more than selecting a model: data must be captured and labeled, the model must be trained, and its predictions must be connected to an application. Learned perception modules such as Mask R-CNN and PoseCNN [2], [3] rely on labeled training data, while the collection and annotation of large datasets such as ImageNet and BDD100K [4], [5] require substantial human effort. Simulators such as CARLA [6] and Gazebo [7] can generate labels automatically, although differences in geometry and appearance can introduce a sim-to-real gap. Neural scene representations such as NeRF [8] and 3D Gaussian Splatting (GSplat [9])

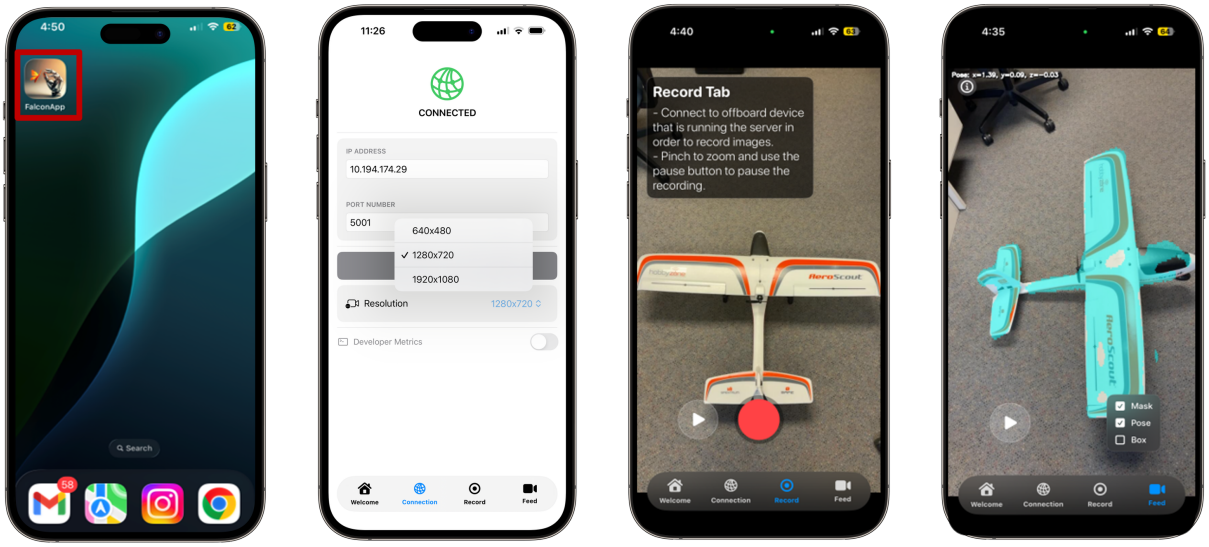

Fig. 2: **FalconApp frontend interface:** the four views, from left to right, show iPhone app installation, TCP connection to the backend, recording mode (Stage 1), and display of offboard inference results (Stage 4) in Figure 1.

support photorealistic rendering and, when object and camera transforms are available, render-aligned supervision. They therefore provide a useful basis for recent robot-learning data engines. This paper focuses on the surrounding *system* problem: connecting mobile capture, photorealistic data generation, model training, and a live inference interface without per-image manual mask or pose annotation.

We present FalconApp, a mobile client–server system for this workflow (Figure 1). A phone frontend records a short handheld capture and uploads it to a backend that reconstructs an editable object GSplat, composites it with diverse backgrounds to synthesize images with aligned masks and 6-DoF poses, and trains an object-specific perception module. For live use, the frontend streams images to the backend for offboard inference and displays the returned mask and pose predictions. FalconApp builds directly on the existing photorealistic rendering and auto-labeling tools in FalconGym 2.0 and the perception module in FalconTrack [1], [10]. Our contribution is not a new renderer or perception backbone. It is the Swift mobile client, frontend–backend protocol, workflow orchestration, and characterization of the integrated capture-to-inference system. In summary, our contributions are: (1) a mobile client–server system that joins object capture, synthetic labeling, training, and offboard live inference in one workflow without per-image manual mask or pose labels; and (2) a characterization across five rigid object instances, including perception accuracy, 12–23 minutes of post-reconstruction data generation and training per object, and a 29–32 ms client–server inference round trip.

II. Related Work

a) Labeled Datasets and Synthetic Data: Large-scale labeled datasets such as ImageNet [4], BDD100K [5], and Ego-Exo4D [11] support research in recognition, driving, and egocentric perception, respectively. Because these are curated corpora, however, they do not themselves provide an on-demand route from a newly captured object to object-specific training data. Simulation platforms such as CARLA [6] and Gazebo [7] provide controllable environments in which data-generation pipelines can be built, but the simulator alone does not specify how a captured object is reconstructed, labeled, used for training, and connected to live inference. FalconApp addresses this infrastructure gap by turning a two-minute object capture into the input to a backend object-onboarding workflow without per-image mask or pose annotation.

b) Photorealistic Simulation and Data Engines: Neural scene representations such as NeRF [8] and 3D Gaussian Splatting [9] enable photorealistic novel-view rendering. When combined with task-specific scene organization, transforms, and rendering logic, these representations can support synthetic-data systems for manipulation [12], driving [13], and visual aerial robotics [14], [15], [16]; the scene representation itself does not produce task labels automatically. FalconGym 2.0 provides editable-GSplat rendering [1], while FalconTrack provides the existing object-level auto-labeling and perception stack [10]. FalconApp builds on these components as a capture-to-live-inference integration layer: it connects phone capture to backend reconstruction, rendering, labeling, training, and offboard inference rather than introducing a new renderer, label generator, or perception model.

c) Object-Centric Perception: Instance segmentation models such as Mask R-CNN [2] and YOLACT [17], together with 6-DoF pose estimators such as PoseCNN [3], PVNet [18], and RGB-D DenseFusion [19], define model architectures for object-centric prediction; supplying suitable object-specific training data remains a separate systems problem. Geometric perspective-n-point (PnP) [20] instead estimates pose from 2D–3D correspondences, whose construction is likewise outside the solver itself. FalconApp does not propose another estimator in this space: it reuses Falcon-

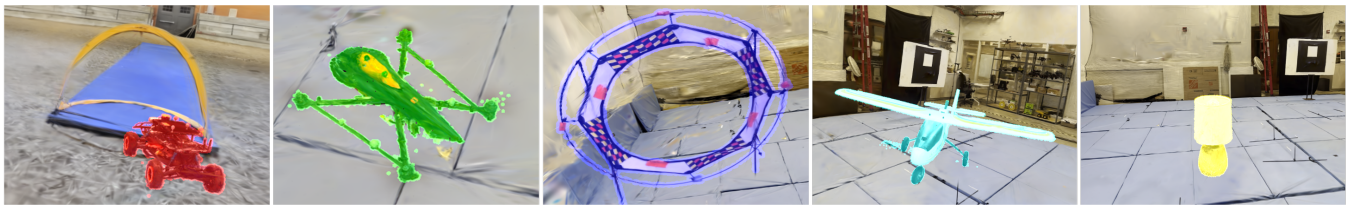

Fig. 3: **Automatically Labeled Synthetic Data:** FalconApp renders synthetic images with render-aligned masks for objects with diverse geometry and appearance.

Track's compact mask-conditioned pose head and connects the auto-generated training data and trained model to a live phone interface through the backend.

d) Capture-to-Inference Systems: Vision-based robot systems increasingly emphasize the infrastructure around the model—data collection, sim-to-real transfer, and reliable hardware-software integration [14], [21], [22]. These systems motivate FalconApp's infrastructure focus, while addressing different robotic tasks and hardware settings. FalconApp's distinct contribution is the integration boundary over FalconGym and FalconTrack: a phone frontend handles capture and live interaction, while a networked backend performs object reconstruction, synthetic-data generation, training, and off-board inference in one capture-to-live-inference workflow.

III. Methodology

FalconApp couples a mobile capture-and-display frontend with a photorealistic reconstruction, auto-labeling, training, and inference backend. A short phone capture of a rigid object is converted into a target-specific perception service for instance segmentation and 6-DoF pose estimation, which the phone accesses over TCP. As summarized in Figure 1, the pipeline has four stages. Stages 1 and 4 provide the phone-facing capture and visualization interfaces; the backend performs reconstruction, data generation, training, and Stage 4 inference. In Section III-A, the mobile frontend records a short handheld video and uploads the capture to the backend. In Section III-B, the backend reconstructs a GSplat asset, isolates the target with FalconGym 2.0's Edit API, and renders synthetic images with masks and poses derived from the simulator state. In Section III-C, the backend trains a perception module from the generated data and packages the resulting model. Finally, in Section III-D, the trained model remains on the backend: the phone streams camera images over TCP and displays the returned predictions.

A. FalconApp Frontend: Recording Mode

FalconApp is implemented in Swift and deployed directly on an iPhone, with the user interface shown in Figure 2. The frontend provides a lightweight capture interface for recording a short RGB video while the user walks around the object to obtain multi-view coverage. Captured video and associated camera metadata are sent to the backend through a TCP client, enabling direct frontend-to-backend upload without manual file transfer. The same mobile interface also displays a welcome page with instructions and the live inference outputs described in Section III-D.

B. FalconApp Backend: Automated Synthetic Data Labeling

After upload, the backend reconstructs a 3D Gaussian Splatting (GSplat) [9] representation of the target from the handheld capture using FalconGym [16]. We then use FalconGym 2.0's Edit API [1] to isolate the object splats from the capture background and obtain an editable object GSplat. To generate training data, the object GSplat is composited with multiple background GSplats and rendered under randomized viewpoints, object scales, and lighting conditions. These randomizations broaden the synthetic appearance and viewpoint coverage used to train the estimator; they do not by themselves eliminate the synthetic-to-real gap. Because FalconGym 2.0 exposes the object transform and camera intrinsics/extrinsics, each synthetic image is paired with a render-aligned foreground mask and the target's relative 6-DoF pose in the camera frame. No user supplies per-image masks or poses. We also include empty-scene images so that training contains examples in which the target is fully out of view. Examples of the generated masks are shown in Figure 3. Rendering and label generation take about 0.2 s per $1280 \times 720 \times 3$ RGB image on an RTX 4090 GPU. For the 1000-image datasets used in Section IV-A, the measured auto-labeling totals are 2–8 minutes per object. GSplat reconstruction and object isolation are required preceding steps, but their time is not reported separately in the current evaluation.

C. FalconApp Backend: Perception Module Training

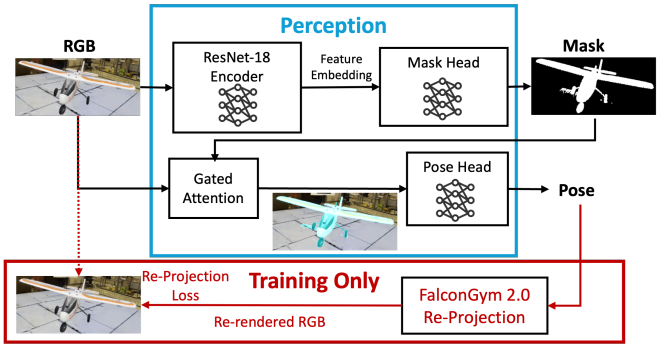

Fig. 4: **Perception Architecture:** an encoder extracts features that feed into a mask prediction head; the predicted mask conditions a gated-attention pose head that regresses the relative 6-DoF target pose. During training, FalconGym 2.0 re-renders the target at the predicted pose to provide a reprojection-consistency term.

Each uploaded object receives its own perception module.

TABLE I: Object-GSplat appearance metrics, post-reconstruction backend time, and live client–server latency. Reconstruction and object-isolation time are excluded.

Object GSplat Appearance					Post-Reconstruction Processing and Live Service			
Object	# Gaussians	SSIM↑	PSNR↑	LPIPS↓	# Synthetic Images	Auto-Label Time	Training Time	Round-Trip Latency
Car	20 k	0.88	31.8	0.09	1000	2 min	10 min	30 ms
Quadrotor	23 k	0.85	32.4	0.06	1000	2 min	10 min	32 ms
Gate	74 k	0.71	30.9	0.12	1000	8 min	15 min	30 ms
Plane	38 k	0.73	30.0	0.08	1000	5 min	15 min	31 ms
Lamp	16 k	0.77	30.7	0.08	1000	2 min	10 min	29 ms

Readers are referred to FalconTrack [10] for details. As shown in Figure 4, the module first predicts a foreground mask, then applies gated attention over the mask and RGB features before feeding them into a pose head to suppress background clutter while preserving appearance cues needed for orientation. Training follows a short staged schedule: we first optimize mask prediction, and then jointly optimize mask and pose using the synthetic RGB, mask, and pose labels. The final objective combines a segmentation loss, pose regression loss, and a reprojection-consistency term that re-renders the object in FalconGym 2.0 at the predicted pose and compares it with the input image using SSIM. Pose and reprojection losses are applied only when the object is in view. Training takes 10–15 minutes per object in the reported runs, after which the model remains on the backend for inference.

D. FalconApp Frontend: Live Offboard Inference

Once training completes, the backend retains the target-specific model and exposes inference to the mobile client over TCP. The frontend streams incoming camera images to the backend, which runs segmentation and 6-DoF pose prediction and returns the results. Each frame is resized to the training resolution before the forward pass, and the app visualizes the returned mask together with the relative pose estimate, as shown in the rightmost subfigure of Figure 2. The measured 29–32 ms latency is therefore a round-trip quantity that includes both backend inference and TCP communication. The current implementation is an offboard inference path and requires a network connection; it does not execute the trained model on the phone.

IV. Experiments

We characterize FalconApp on five rigid object instances that span different geometry, scale, and symmetry: a car, a quadrotor, a gate, a plane, and a lamp. Our evaluation separates the available measurements into backend processing time, live client–server latency, and perception accuracy. Because each target receives its own model, these experiments evaluate instance onboarding rather than category-level generalization.

A. Pipeline Runtime and Reconstruction Quality

We first report descriptive appearance metrics for the object GSplats reconstructed in Section III-B, together with the measured data-generation, training, and client–server inference times. For reconstruction quality, we report the standard photorealism metrics SSIM, PSNR, and LPIPS. We then report the number of synthetic images, automatic labeling time, training time, and round-trip latency for each object. Reconstruction, object-isolation, and model-loading times were not recorded separately and are therefore not included in the post-reconstruction processing totals.

Across the five reconstructed assets in Table I, SSIM ranges from 0.71 to 0.88, PSNR from 30.0 to 32.4, and LPIPS from 0.06 to 0.12. Generating 1000 synthetic images and training one perception module takes 12–23 minutes after reconstruction (15.8 minutes on average). The 29–32 ms latency includes backend inference and TCP communication with the phone; it measures the offboard client–server loop rather than local phone inference. This rate supports roughly 30 live updates per second under the reported setup, but does not by itself establish robustness to camera motion or network changes.

B. Perception Module Accuracy

We next evaluate FalconApp's perception module on the same five objects against two pose-estimation baselines: a geometric perspective-n-point (PnP) method [20] and a learned neural pose estimator (NPE) [16]. PnP is a standard instance-specific 6-DoF estimator for known rigid objects, but it relies on manually selected correspondences rather than an auto-labeling pipeline. NPE is a representative learned regressor trained on auto-labeled synthetic data, but it must be trained separately *per object and per background*, whereas FalconApp trains one module per object that generalizes across backgrounds through domain randomization. The NPE baseline is available only for the three objects (car, quadrotor, gate) shared with prior work [10]; it was not trained for the plane or lamp, and those cells are marked "–". We report mean translational error (MTE, normalized by object size as a percentage) and mean angular error (MAE, in radians). For completeness, FalconApp's simulated mask IoU is 0.83, 0.79, 0.72, 0.81, and 0.62 for the car, quadrotor, gate, plane, and lamp, respectively; real-image IoU is unavailable because the evaluation set does not contain manually annotated masks.

We evaluate all methods on 200 unseen FalconGym 2.0 images per object drawn across two environments, and then on 200 real images per object drawn across two real environments. An ArUco marker-based calibration procedure provides the real-world camera-frame pose used for evaluation; the markers are used only to obtain evaluation ground

TABLE II: Pose-estimation accuracy on five objects in FalconGym 2.0 and the real world. MTE and MAE are lower-is-better; bold denotes the best available result for each object and metric.

			FalconGym 2.0 (2 Environments)		**Real World (2 Environments)**	
Object	Method	Per-Object Setup	MTE (%)↓	MAE (rad)↓	MTE (%)↓	MAE (rad)↓
Car	PnP [20]	Manual corr.	52%	0.88	82%	1.21
	NPE [16]	Auto labels	42%	0.54	52%	0.77
	FalconApp	Auto labels	**24%**	**0.27**	**43%**	**0.38**
Quadrotor	PnP [20]	Manual corr.	71%	1.12	82%	0.82
	NPE [16]	Auto labels	54%	0.52	61%	0.51
	FalconApp	Auto labels	**32%**	**0.31**	**41%**	**0.37**
Gate	PnP [20]	Manual corr.	60%	0.76	72%	0.69
	NPE [16]	Auto labels	54%	0.74	68%	0.74
	FalconApp	Auto labels	**41%**	**0.41**	**52%**	**0.52**
Plane	PnP [20]	Manual corr.	37%	0.62	57%	0.66
	NPE [16]	Auto labels		–		–
	FalconApp	Auto labels	**24%**	**0.32**	**34%**	**0.44**
Lamp	PnP [20]	Manual corr.	**62%**	0.98	**72%**	**1.1**
	NPE [16]	Auto labels		–		–
	FalconApp	Auto labels	71%	**0.84**	85%	1.4

truth, not for onboarding or inference.

FalconApp has lower MTE and MAE than PnP for the car, quadrotor, gate, and plane in both simulation and the real world. On the three objects for which NPE is available, FalconApp also has lower MTE and MAE under the reported protocol. NPE was not available for the plane or lamp, so this comparison is limited to the three shared objects rather than a comprehensive learned-baseline study. The lamp is a clear failure case: although FalconApp has lower simulated MAE than PnP, its simulated MTE and both real-world errors are worse. Its rotational symmetry makes a unique orientation difficult to identify, and no symmetry-aware metric is reported here; the lamp results should therefore be interpreted as exposing a limitation rather than graceful degradation.

C. Human and Compute Cost

The main systems benefit is the separation of per-object capture effort from per-image annotation effort. Each new object requires a two-minute handheld capture, so both capture effort and backend compute still scale linearly with the number of objects. After reconstruction, producing the 1000 rendered examples and training the model takes 12–23 minutes on the backend, while the rendered masks and poses require no per-image manual annotation. We do not quantify an annotation-time counterfactual because no manual-labeling study was performed. Accordingly, the evidence supports a narrower claim: FalconApp replaces per-image mask and pose labeling with a fixed capture step plus per-object backend computation; it does not demonstrate constant total human effort or fleet-scale throughput.

V. Conclusion & Discussion

We presented FalconApp, a mobile client–server system that connects a two-minute object capture to synthetic labeling, object-specific model training, and live offboard inference without per-image manual mask or pose labels. The system reuses FalconGym 2.0 and FalconTrack for rendering, labeling, and perception, while contributing the mobile client, protocol, orchestration, and integrated-system characterization. Across the five evaluated rigid object instances, post-reconstruction data generation and training take 12–23 minutes per object, and the measured client–server inference round trip is 29–32 ms.

The evidence is limited to five known, rigid object instances captured one at a time; it does not establish category-level generalization, articulated-object support, multi-object onboarding, or usability by non-expert users. The learned model is object-specific, and its accuracy continues to depend on synthetic-to-real transfer. The rotationally symmetric lamp is the clearest failure case: its simulated translation error and both real-world errors exceed PnP, while only its simulated angular error is lower. Because no symmetry-aware metric is reported, these results expose unresolved orientation ambiguity rather than establishing performance on symmetric shapes. The learned NPE comparison is available for only three objects, and the reported round-trip latency is specific to the evaluated client–server configuration. Finally, reconstruction, training, and live inference currently depend on the GPU backend. Future work will export an optimized perception model for local phone inference, removing the network dependency during live use. Additional directions include symmetry-aware pose representations and extensions to articulated and multi-object captures.

References

[1] Y. Miao, E. Yuceel, G. Fainekos *et al.*, "Performance-guided refinement for visual aerial navigation using editable gaussian splatting in falcongym 2.0," in *Proceedings of IEEE International Conference on Robotics and Automation (ICRA)*, 2026.

[2] K. He, G. Gkioxari, P. Dollár *et al.*, "Mask r-cnn," in *2017 IEEE International Conference on Computer Vision (ICCV)*, 2017, pp. 2980–2988.

[3] Y. Xiang, T. Schmidt, V. Narayanan *et al.*, "Posecnn: A convolutional neural network for 6d object pose estimation in cluttered scenes," in *Robotics: Science and Systems (RSS)*, 2018. [Online]. Available: https://github.com/yuxng/PoseCNN

[4] J. Deng, W. Dong, R. Socher *et al.*, "Imagenet: A large-scale hierarchical image database," in *2009 IEEE Conference on Computer Vision and Pattern Recognition*, 2009, pp. 248–255.

[5] F. Yu, H. Chen, X. Wang *et al.*, "Bdd100k: A diverse driving dataset for heterogeneous multitask learning," in *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2020.

[6] A. Dosovitskiy, G. Ros, F. Codevilla *et al.*, "CARLA: An open urban driving simulator," in *Proceedings of the 1st Annual Conference on Robot Learning*, 2017, pp. 1–16.

[7] N. Koenig and A. Howard, "Design and use paradigms for gazebo, an open-source multi-robot simulator," in *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) (IEEE Cat. No.04CH37566)*, vol. 3, 2004, pp. 2149–2154 vol.3.

[8] B. Mildenhall, P. P. Srinivasan, M. Tancik *et al.*, "Nerf: representing scenes as neural radiance fields for view synthesis," *Commun. ACM*, vol. 65, no. 1, p. 99–106, Dec. 2021. [Online]. Available: https://doi.org/10.1145/3503250

[9] B. Kerbl, G. Kopanas, T. Leimkühler *et al.*, "3d gaussian splatting for real-time radiance field rendering," *ACM Transactions on Graphics*, vol. 42, no. 4, July 2023. [Online]. Available: https://repo-sam.inria.fr/fungraph/3d-gaussian-splatting/

[10] Y. Miao, K. Gandiboyina, N. Giles *et al.*, "Falcontrack: Photorealistic auto-labeled perception and physics-aware vision-based aerial tracking," 2026. [Online]. Available: https://arxiv.org/abs/2606.29783

[11] K. Grauman, A. Westbury, L. Torresani *et al.*, "Ego-exo4d: Understanding skilled human activity from first- and third-person perspectives," *2024 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 19 383–19 400, 2023. [Online]. Available: https://api.semanticscholar.org/CorpusID:265506384

[12] S. Yang, W. Yu, J. Zeng *et al.*, "Novel demonstration generation with gaussian splatting enables robust one-shot manipulation," *arXiv preprint arXiv:2504.13175*, 2025.

[13] Y. Yan, H. Lin, C. Zhou *et al.*, "Street gaussians: Modeling dynamic urban scenes with gaussian splatting," in *ECCV*, 2024.

[14] J. Low, M. Adang, J. Yu *et al.*, "Sous vide: Cooking visual drone navigation policies in a gaussian splatting vacuum," *IEEE Robotics and Automation Letters (under review)*, 2024, available on arXiv: https://arxiv.org/abs/2412.16346.

[15] Q. Chen, J. Sun, N. Gao *et al.*, "Grad-nav: Efficiently learning visual drone navigation with gaussian radiance fields and differentiable dynamics," 2025. [Online]. Available: https://arxiv.org/abs/2503.03984

[16] Y. Miao, W. Shen, and S. Mitra, "Falcongym: A photorealistic simulation framework for zero-shot sim-to-real vision-based quadrotor navigation," in *2025 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2025, pp. 17 154–17 161.

[17] D. Bolya, C. Zhou, F. Xiao *et al.*, "Yolact++ better real-time instance segmentation," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 44, no. 2, pp. 1108–1121, Feb. 2022. [Online]. Available: https://doi.org/10.1109/TPAMI.2020.3014297

[18] S. Peng, Y. Liu, Q. Huang *et al.*, "Pvnet: Pixel-wise voting network for 6dof pose estimation," in *CVPR*, 2019.

[19] C. Wang, D. Xu, Y. Zhu *et al.*, "Densefusion: 6d object pose estimation by iterative dense fusion," in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 2019, pp. 3343–3352.

[20] E. Marchand, H. Uchiyama, and F. Spindler, "Pose estimation for augmented reality: A hands-on survey," *IEEE Transactions on Visualization and Computer Graphics*, vol. 22, no. 12, pp. 2633–2651, 2016.

[21] I. Geles, L. Bauersfeld, A. Romero *et al.*, "Demonstrating agile flight from pixels without state estimation," in *Robotics: Science and Systems XX, Delft, The Netherlands, July 15-19, 2024*, D. Kulic, G. Venture, K. E. Bekris *et al.*, Eds., 2024. [Online]. Available: https://doi.org/10.15607/RSS.2024.XX.082

[22] E. Kaufmann, L. Bauersfeld, A. Loquercio *et al.*, "Champion-level drone racing using deep reinforcement learning," *Nature*, vol. 620, no. 7976, pp. 982–987, 2023.