Performance-Guided Refinement for Visual Aerial Navigation using Editable Gaussian Splatting in FalconGym 2.0

Yan Miao[1], Ege Yuceel[1], Georgios Fainekos[2], Bardh Hoxha[2], Hideki Okamoto[2] and Sayan Mitra[1]

***Abstract*— Visual policy design is crucial for aerial navigation. However, state-of-the-art visual policies often overfit to a single track and their performance degrades when track geometry changes. We develop *FalconGym 2.0*, a photorealistic simulation framework built on Gaussian Splatting (GSplat) with an *Edit API* that programmatically generates diverse static and dynamic tracks in milliseconds. Leveraging FalconGym 2.0's editability, we propose a *Performance-Guided Refinement (PGR)* algorithm, which concentrates visual policy's training on challenging tracks while iteratively improving its performance. Across two case studies (fixed-wing UAVs and quadrotors) with distinct dynamics and environments, we show that a single visual policy trained with PGR in FalconGym 2.0 outperforms state-of-the-art baselines in generalization and robustness: it generalizes to three unseen tracks with 100% success *without* per-track retraining and maintains higher success rates under gate-pose perturbations. Finally, we demonstrate that the visual policy trained with PGR in FalconGym 2.0 can be *zero-shot sim-to-real transferred* to a quadrotor hardware, achieving a 98.6% success rate (69 / 70 gates) over 30 trials spanning two three-gate tracks and a moving-gate track.**

I. Introduction

Visual aerial navigation is critical for applications such as mapping, search-and-rescue, environmental monitoring and racing. Recent progress in photorealistic simulation environments has fueled zero-shot sim-to-real success for visual aerial navigation. Notably, SOUS VIDE [1] used Gaussian Splatting (GSplat) [2] to reconstruct an indoor lab and achieved zero-shot sim-to-real navigation; FalconGym [3] used NeRF [4] to build digital twins of three racing tracks and demonstrated zero-shot sim-to-real transfer of quadrotor gate crossing via imitation learning; Geles *et al.* [5] reported strong sim-to-real performance with a vision-only asynchronous actor-critic on three quadrotor racing tracks. While effective, these benchmarks face generalization limits: all three achieve high performance on their training tracks but do not generalize to unseen tracks (as we also confirm in Section IV), restricting broader applicability. GRaD-Nav [6] improves robustness to different gate positions and background distractors by using one policy, but still relies on constructing multiple GSplat tracks first and training in those.

In this paper, we focus on developing a single visual policy that can traverse a family of tracks, where each track consists of an ordered sequence of tight racing gates, as shown in Figure 1 and Figure 4. To enable such cross-track generalization from limited real data, we develop *FalconGym 2.0*. From just minutes of real-world videos, FalconGym 2.0 can generate arbitrarily many synthetic yet photorealistic tracks

[1]University of Illinois at Urbana-Champaign
[2]Toyota Research Institute of North America

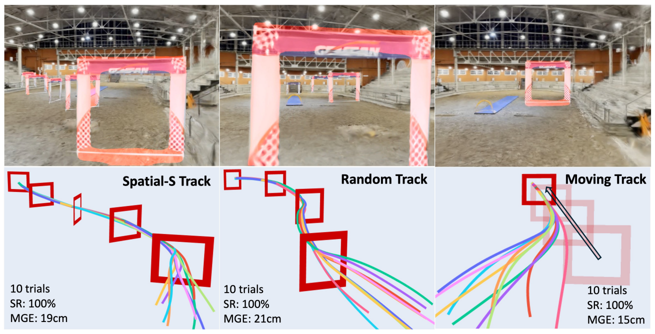

Fig. 1: **Trajectories for UAV case study in FalconGym 2.0 across three unseen tracks (Spatial-S, Random and Moving).** Top row: red overlays visualize predicted gate masks (Section III-B). Bottom row: 10 trials per track from different initial states; the translucent gates in the Moving track show the gate's past positions.

for training without additional real-world data collection. This is achieved by our *Edit API*, as shown in Figure 3, that allows programmatic editing of gates in the GSplat to create diverse tracks in milliseconds. Beyond static edits, our Edit API also supports 4D time-varying simulation with moving gates, enabling dynamic tracks in Section IV. Although this paper focuses on aerial navigation and gate-editing, the same Edit API could readily extend to broader robotics settings (e.g., obstacle placement for ground robots).

FalconGym 2.0's editable capability to generate arbitrarily many photorealistic tracks unlocks a range of visual policy training strategy from active learning to curriculum learning [7]. We develop a *Performance-Guided Refinement (PGR)* algorithm that: (i) identifies challenging tracks where the visual policy underperforms; (ii) synthesizes similar yet slightly different challenging tracks with the Edit API to augment the training dataset; and (iii) iteratively refines the visual policy via imitation learning from a state-based expert.

Beyond the Edit API and the PGR algorithm it enables, FalconGym 2.0 improves over FalconGym [3] via: (i) replacing NeRF rendering with fast GSplat to accelerate training; (ii) eliminating expensive motion capture with an accessible ArUco marker for world-frame simulation reconstruction.

Through two case studies (fixed-wing UAVs and quadrotors) with different dynamics, environments and gate geometries, we show that our visual policy trained with PGR can generalize to three unseen tracks in FalconGym 2.0 with 100% success. Moreover, we outperform state-of-the-art visual baselines [3], [5] in both generalization and robustness: our single visual policy operates across three unseen tracks, where baselines require separate per-track policies, and we maintain higher performance under gate-pose perturbations. Finally, the visual policy trained with PGR in FalconGym 2.0 can zero-shot transfer to a quadrotor hardware, achieving 98.6% (69 / 70 gates) success rate in 30 hardware trials spanning two 3-gate tracks and a moving-gate track, as shown in Figure 6.

In summary, our contributions are: (1) *FalconGym 2.0*: an editable photorealistic simulation framework based on GSplat that supports fast world-frame modification of environment configurations. (2) *Performance-Guided Refinement (PGR)*: an algorithm that leverages FalconGym 2.0's editability to expose the visual policy's training to challenging tracks and iteratively improve performance. (3) *Zero-Shot Sim-to-real*: Our visual policy trained with PGR in FalconGym 2.0 can be zero-shot transferred to a real quadrotor hardware to traverse three unseen tracks.

II. Related Work

a) Sim-to-real in robotics.: Sim-to-real transfer is a longstanding goal in robotics, due to its efficient and safe training before deployment. With advances in photorealistic scene reconstruction like NeRF [4] and 3D Gaussian Splatting (GSplat) [2], recent robotics work increasingly trains in high-fidelity simulators. NeRF2Real [8] and RialTo [9] demonstrate NeRF-to-real transfer for humanoid navigation and robot manipulation, respectively. Vid2Sim [10] converts real-world videos into interactive simulators for urban navigation using GSplat reconstruction, while RoboSplat [11] and Splat-MOVER [12] leverages GSplat for domain randomization to improve robot manipulation performance.

In aerial navigation, FalconGym [3] constructs NeRF-based digital twins of racing tracks and achieves zero-shot sim-to-real quadrotor gate crossing via imitation learning, while SOUS VIDE [1] uses GSplat to build an indoor digital twin and also reports zero-shot transfer. Geles *et al.* [5] train a vision-only policy using async actor-critic and demonstrate high-speed racing ($40\,\mathrm{km/h}$) across three tracks. Despite strong sim-to-real results, these benchmarks generally require per-track training and struggle to generalize to unseen tracks.

b) Policy Refinement by Environment Shaping: Curriculum-based environment shaping [7] has been used to improve control policy in robotics by adaptively shaping training environments. It has been successful in simulation for RL agents in bipedal walkers [13]. [14] uses a similar "environment policy" idea for quadrotor navigation. However, these work used state-based control and did not involve rendering or visual policy, while our PGR algorithm adapts an editable photorealistic GSplat environment to improve visual policy performance for aerial navigation.

c) Editable Gaussian Splatting: Recent work focuses on using editable GSplat to generate synthetic but photorealistic environment. GaussianEditor [15] enables text-guided appearance and geometry edits of objects in GSplat. VCEDIT [16] ensures multi-view consistency when using diffusion-guided GSplat edits. Instruct-GS2GS [17] provides instruction-based editing of GSplat scenes via 2D diffusion. While these work advance GSplats scene editing in computer vision, they focus on improving photorealism and accuracy, rather than connecting to downstream closed-loop robotics applications. In contrast, FalconGym 2.0 not only provides open-source Edit APIs to edit objects in GSplat, but also couples this editability with a performance-guided refinement algorithm that iteratively improves downstream visual policy's performance in the closed loop.

III. Methods

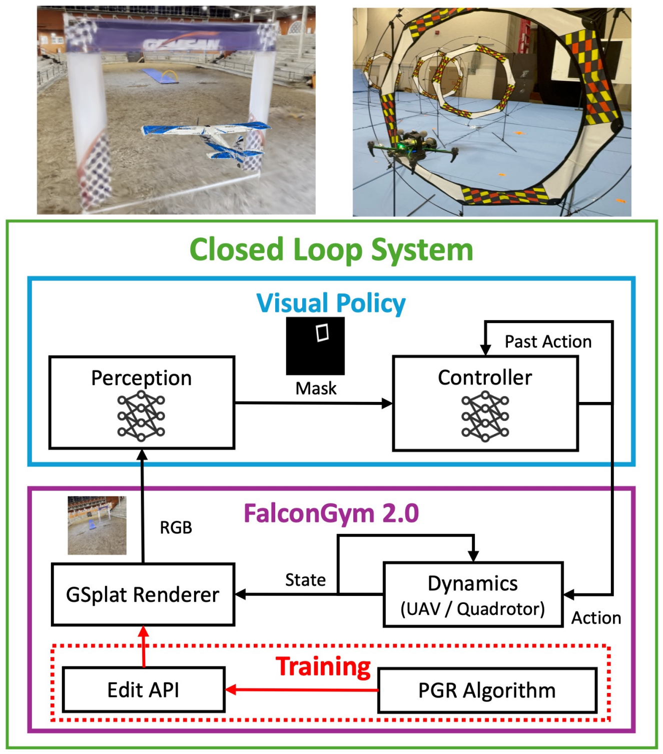

Fig. 2: **Closed-loop system in FalconGym 2.0:** we provide dynamics for a fixed-wing UAV and a quadrotor, and a GSplat renderer that produces photorealistic RGB from arbitrary camera poses in either scene. At each timestep, the dynamics propagate the state, the renderer generates an RGB image, a perception module predicts a gate mask, and a controller consumes the mask plus past actions to predict the next action. During training, a *Performance-Guided Refinement (PGR)* algorithm (Section III-C) focuses training on challenging tracks generated using *Edit API* (Section III-A)

In this paper, we consider visual aerial navigation in the gate-based track setting, where each track is an ordered sequence of tight racing gates and the aerial vehicle must safely traverse a set of tracks using only visual feedback. Successful trajectories are shown in FalconGym 2.0 for a fixed-wing UAV (Figure 1) and a quadrotor (Figure 4), and on hardware for the quadrotor (Figure 6).

Prior work [3], [5] tackles this same aerial navigation problem and reports strong gate-crossing performance; however, these methods require *per-track* training and a visual policy trained on one track does not generalize reliably to unseen tracks. This overfitting could be costly in practice, since recollecting data and retraining for every new track is time-consuming. Therefore, our goal is to develop a single visual policy that operates zero-shot on unseen tracks, eliminating

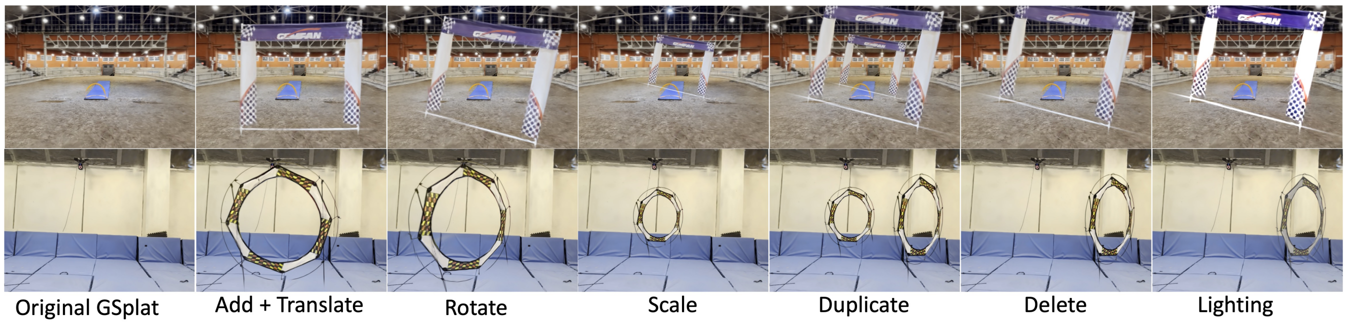

Fig. 3: **Edit API in FalconGym 2.0.** Our Edit API (Section III-A) provides world-frame programmatic placement of objects while the backend handles all coordinates and camera-to-world transform. The seven API: `add`, `translate`, `rotate`, `scale`, `duplicate`, `delete`, and `lighting`, allow users to modify object pose, size, and appearance to generate a photorealistic 4D simulation environment. Shown are gates edits across two environments. This editable capability enables PGR algorithm to improve visual policy (Section III-C).

per-track retraining while maintaining robustness to track-configuration changes. In this paper, we restrict the tracks to be dynamically feasible and observable (when the aerial vehicle crosses current gate center orthogonally, the next gate must be within the camera's field of view). This observability requirement that keeps the next gate in sight is reasonable in a visual-only setting without knowledge of the map.

Our method comprises three components. First, we present *FalconGym 2.0* (Sec.III-A), a GSplat-based photorealistic simulation framework coupled with an *Edit API* that enables dynamic gate placement, improving on [1], [3] static simulation. Second, we design a visual policy (Section III-B) in the closed loop in FalconGym 2.0 that mitigates the overfitting observed in earlier work [3] through modular architecture that separates perception and control. Third, we introduce a novel *Performance-Guided Refinement (PGR)* algorithm (Section III-C) that uses FalconGym 2.0's editability to expose visual policy to challenging tracks and iteratively refine its performance.

A. *FalconGym 2.0: Editable GSplat*

We propose a photorealistic simulation framework *FalconGym 2.0*, that is capable of developing and testing different visual policies, while improving on FalconGym [3] by replacing NeRF with GSplat for fast rendering; substituting motion-capture with an ArUco marker for world-frame alignment. [1] More importantly, FalconGym 2.0 introduces an *Edit API* that's capable of generating arbitrarily many training tracks.

a) GSplat Scene representation: A trained GSplat scene consists of a set of N anisotropic Gaussians, i.e. $\mathcal{S} = \{ (\mu_j, \Sigma_j, c_j, \alpha_j) \}_{j=1}^{N}$, where $\mu_j \in \mathbb{R}^3$ is the mean; $\Sigma_j \in \mathbb{R}^{3\times3}$ is the covariance, parameterized via a rotation $R_j \in \mathrm{SO}(3)$ and per-axis scales $s_j \in \mathbb{R}^3_{>0}$ (i.e., $\Sigma_j = R_j \operatorname{diag}(s_j^2) R_j^\top$); $c_j \in \mathbb{R}^3$ is the color; and $\alpha_j \in [0,1]$ is the opacity. For rendering an image as seen by a camera in this scene S, the N 3D Gaussians are projected onto the camera's image plane then their colors are blended according to their depths relative to the camera.

b) Edit API: Because NeRFStudio [20] (the library we used to train GSplat) optimizes scenes in an internal coordinate frame, programmatic editing is inconvenient for users who reason in a world frame. FalconGym 2.0 instead exposes world-frame edits and handles all coordinate conversions and camera-to-world transformations in the backend. To edit a scene in FalconGym 2.0, users can first select an object either via predefined Gaussian IDs (e.g., gate primitives from another GSplat scene provided by us) or via user-defined world-frame bounding boxes, then apply pose, color and scale edits via our API. Concretely, we provide seven composable operations, as shown in Figure 3: `add()` inserts objects from another GSplat scene; `translate()` moves selected objects to user-defined poses; `rotate()` rotates the selected objects around its center for user-defined rotation; `scale()` scales the selected objects' sizes; `duplicate()` clones the selected object; `delete()` removes selected objects; `lighting()` adjusts gaussian-level colors for selected objects. Each operation updates the selected objects' corresponding Gaussians' μ_j, Σ_j (via R_j, s_j), c_j, and α_j on the backend. Since each API is a tensor operation, on average each operation finishes ~ 0.004s with a 4090 GPU. Users can freely combine any of the 7 API to accomplish complex edits and build customized 4D simulations. (e.g., moving gates in Section IV). Although our experiments focus on aerial navigation with gates, we expect the same Edit API to work on broad robotics tasks that would benefit from photorealistic editable GSplat scenes.

B. *Visual Policy in the Closed Loop*

In this subsection, we introduce our closed-loop system architecture and visual policy design, as shown in Figure 2. The previous subsection has introduced the Edit API for

[1]More specifically, to construct FalconGym 2.0, a human operator uses the onboard camera to capture a 3-minute video across the flying arena from diverse viewpoints. We recover camera intrinsics and initial poses with COLMAP [18]. Because COLMAP's frame is arbitrary, we align it to a physically meaningful world frame by placing an easily accessible ArUco marker in the scene, thereby eliminating the need for expensive motion capture. Using OpenCV's ArUco detector, we locate the marker center in a subset of images where the marker is visible and treat it as the global origin. We then compute the rigid transform between the COLMAP and world frames via the Kabsch-Umeyama method [19]. Finally, the images and poses are fed to the NeRFStudio `Splatfacto` pipeline [20] to train a photorealistic GSplat scene in world coordinates.

moving objects (i.e. gates) within GSplat scenes; here we focus on moving the aerial vehicle (i.e., the onboard camera). Although the aerial vehicle could be placed in arbitrary poses in FalconGym 2.0, closed-loop control requires its motion obey physically meaningful dynamics. FalconGym 2.0 provides two aerial dynamics models: a Dubins airplane dynamics model for fixed-wing UAVs [21] and a quadrotor dynamics model [22]. The dynamics model is implemented in a plug-and-play fashion so users can substitute it with alternative platform's dynamics (e.g., VTOL or ground robots).

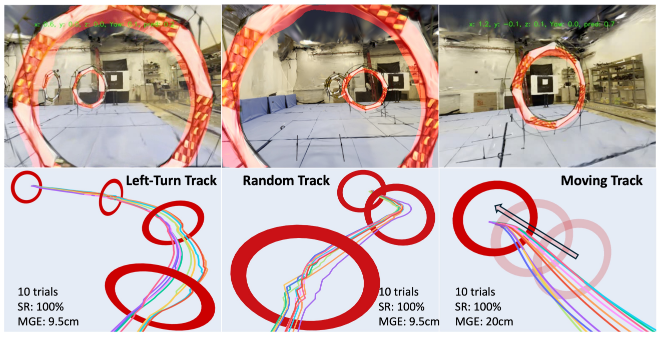

Fig. 4: Trajectories for quadrotor case study in FalconGym 2.0 across three unseen tracks (Left-Turn, Random and Moving).

With renderer and dynamics in place, next we focus on the visual policy design. [3] is quite successful in zero-shot sim-to-real quadrotor navigation by an end-to-end ViT policy. However, as also acknowledged by their authors, their visual policy trained on one track tends to specialize to that track. Our further experiments confirm [3]'s overfitting: even after removing the gates entirely using our Edit API, [3]'s visual policy continued to fly around the memorized route, suggesting reliance on background appearance rather than gate-relevant features. Moreover, [3] has to run the hardware experiments offboard due to a heavy dual-Vision-Transformer (ViT) policy, which also required ground-truth state of the next gate.

Therefore, to mitigate overfitting and reduce model complexity for onboard deployment, we design a modular architecture that decouples perception from control, inspired by [5] which trains a RL algorithm based on gate-perception masks to fly through racing gates. First, we design a perception module that predicts a gate mask. Next, a lightweight controller consumes this mask with a short history of past controls to output the next action, as shown in Figure 2. As validated in Section IV, this modular pipeline not only improves generalization, but also supports on-board execution due to the smaller model size compared to [3]'s dual-ViT setup.

a) Gate-Detection Perception Module: We train the perception module completely in FalconGym 2.0. As shown in the blue box in Figure 2, the perception module takes an onboard RGB image and predicts a binary mask where white pixels indicate gates and black pixels indicate background. We adopt a U-Net [23] backbone for gate segmentation for its strong performance on dense prediction. To obtain ground-truth masks, we automatically generate them analytically via standard 3D-to-2D projection techniques in computer vision. Because gate positions (placed in FalconGym 2.0 through our Edit API) are known and the geometry (e.g. diameter) is measured, we can project 3D gates to a 2D image plane using known camera matrix. Pixels whose coordinates lie within the gate's projected ring (i.e., between inner and outer boundaries) are labeled white, and all others black.

To collect training data for the U-Net, we: (i) leverage the Edit API to place a two-gate track in the workspace; (ii) sample a feasible camera pose at an appropriate distance with yaw roughly oriented toward the track; and (iii) render the RGB image and compute its ground truth 2D mask as above. We sample two-gate tracks because we want the U-Net to learn about scenarios where one gate might be occluded by the other, and experiments show our U-Net could generalize to multi-gate detection. We gather RGB images (square-gate UMX for UAV case study and circular-gate for quadrotor case study) and train separate U-Net for each case study with supervised learning on these image-mask pairs. Qualitative results of the U-Net are shown in Figure 1 and 4 in FalconGym 2.0, while Figure 6 visualizes its performance when zero-shot deployed on hardware.

b) Controller: With a predicted mask, the controller receives an explicit geometry-focused signal of gate locations, potentially reducing reliance on background cues and mitigating overfitting. Next, we further improve the architecture of [3] by removing IMU inputs and instead feeding a short history of past controls to the controller (blue box in Figure 2), which provides implicit temporal context. Next, the controller training follows a similar imitation learning procedure as in [3]: we first implement a state-based expert that flies through different tracks in simulation; at each timestep, we render the onboard RGB image and record the state-based controller's expert action. The RGB image is passed through the trained U-Net to obtain a binary mask, and we form supervised pairs where the masked image coupled with the past control actions are used to predict the current action to train the controller.

Thanks to the Edit API, now we can synthesize essentially arbitrarily many tracks in FalconGym 2.0 to train both perception and controller without additional per-track real-world effort required by [1], [3], [5]. To sample efficiently, our unique design choice is to train on *two-gate tracks*. Intuitively, the initial state together with two successive gates spans the local geometric variability of longer courses; a controller that performs well on such segments could generalize well to multi-gate tracks by invariance and composition, as is empirically confirmed in Section IV.

C. Performance-Guided Refinement Training

A straightforward method to collect training data for the visual policy would be to uniformly sample the two-gate track space that is dynamically feasible and observable (as defined at the start of this section). However, uniform sampling can be sample-inefficient in a large high-dimensional workspace. With our Edit API, we can steer training data collection toward the visual policy's weak spots and iteratively refine to improve the visual policy.

Inspired by adversarial training [24] and min-max problem [25] which tries to minimize the possible loss for a worst case (maximum loss) scenario, we cast our aerial navigation as a min-max problem:

$$\min_{\theta} \max_{g \in G} \mathbb{E}_{\tau \sim \pi_\theta} \big[\mathcal{L}(\tau; g) \big], \tag{1}$$

where g parameterizes a two-gate track, G is the dynamically feasible and observable two-gate space, and $\mathcal{L}$ measures task performance on a trajectory rollout τ achieved by the visual policy (perception and controller) π parameterized by θ. We define the task performance function as $\mathcal{L}(\tau; g) = \mathbf{1}\{\text{collision or timeout}\} + \lambda_{\text{pos}} \big\| p' - c(g) \big\|_2$, where $c(g) \in \mathbb{R}^3$ is the gate center, and $p' \in \mathbb{R}^3$ is the aerial vehicle's position when crossing the target gate plane. The second term is only evaluated on successful crossings. Intuitively, "maximizer" proposes challenging gate placements to cause visual policy π_θ's poor performance, while "minimizer" updates π_θ to reduce loss to improve performance. This way, training is focused on where visual policy under-performs.

Algorithm 1 Performance-Guided Refinement of Visual Policy

```
Input: Gate space $G$ partitioned into $M$ grids, state-based
    expert policy $\pi_s$, iterations $T$, validation gate set $G_{\text{val}}$
Output: Trained visual policy $\pi_\theta$
 1: $\mathcal{D} \leftarrow \varnothing$                              ▷ Training dataset
 2: Sample initial tracks $G_1 \subset G$ uniformly at random
 3: for $t = 1$ to $T$ do
 4:     for all two-gate track $g \in G_t$ do
 5:         Run closed-loop trajectory rollout with $\pi_s$ and $g$
 6:         Collect trajectory (action, image) and add to $\mathcal{D}$
 7:     end for
 8:     Train visual policy $\pi_\theta$ on $\mathcal{D}$
 9:     for $i = 1$ to $M$ do     ▷ evaluate grid-wise performance
10:         $\ell_i \leftarrow \frac{1}{|G_{\text{val}} \cap M_i|} \sum_{g \in G_{\text{val}} \cap M_i} \mathcal{L}(\pi_\theta; g)$
11:     end for
12:     Compute normalized grid weights:
13:     $w_i \leftarrow \frac{\ell_i}{\sum_{i=1}^{M} \ell_i} \quad \forall i = 1, \dots, M$
14:     $w_i \leftarrow (1-\beta)\, w_i + \frac{\beta}{M}$              ▷ Avoid Mode Collapse
15:     Generate next set $G_{t+1}$ by: first choosing $M_i$ with prob-
        ability $w_i$ and then sample $g \sim M_i$ uniformly
16: end for
```

Yet directly solving Equation (1) is infeasible, so we implement grid-based *Performance-Guided Refinement (PGR)*, as shown in Algorithm 1. We partition the two-gate space G into M grids $\{M_i\}_{i=1}^{M}$. During the first iteration, we synthesize G_1 by first uniformly sampling the grids and then drawing two-gate layouts uniformly within selected grids. We delete the dynamically infeasible and unobservable tracks automatically using the expert state-based controller and geometric calculation. Next, for each iteration t: we (i) collect trajectory rollouts on a batch of two-gate layouts G_t using the expert state-based controller to augment the perception and imitation dataset $\mathcal{D}$, (ii) train the visual policy π_θ on $\mathcal{D}$, (iii) evaluate per-grid validation performance ℓ_i using a held-out set G_{val}, and (iv) *resample* the next batch G_{t+1} by first drawing grids with probability proportional to ℓ_i and then sampling tracks uniformly within each selected grid. We apply this PGR to both perception and controller training since poor performance could come from both modules. To avoid mode collapse (as observed in [24]), we mix a small fraction of uniform sampling $w_i \leftarrow (1-\beta)\, w_i + \beta/M$, and reuse a fixed G_{val} for stable scoring. "Ours (w/ PGR)" in Table I and Table II corresponds to performance-guided refinement algorithm, while "Ours (w/o PGR)" refers to the approach of uniform sampling two-gate tracks.

IV. Experiments

We evaluate our approach in two case studies: fixed-wing UAVs and quadrotors. For each case study, we first describe the experimental setups, and then quantitatively evaluate our visual policy against baselines in FalconGym 2.0. We further demonstrate zero-shot sim-to-real transfer on the quadrotor case study in Section IV-B.

A. Case Study 1: fixed-wing UAV

We model the fixed-wing UAV with a Dubins Airplane dynamics [21]. The state is (x, y, z, ψ, θ) (3D position, yaw and pitch). The control action is bank rate and pitch rate. Forward speed is fixed at $7\,\mathrm{m/s}$. We implement the state-based controller also from [21], to serve as a baseline and the expert of our imitation learning approach (Section III-B).

Our UAV flying arena measures $40 \times 20 \times 4\,\mathrm{m}$. The square gates used in the UAV case study have an inner side length of $200\,\mathrm{cm}$. The UAV has a width of $40\,\mathrm{cm}$. A gate crossing is deemed *success* if, at the instant the aerial vehicle passes the gate plane, the distance from the UAV to the gate center is less than $80\,\mathrm{cm}$. We evaluate the performance of gate crossing on two metrics: *Success Rate (SR)*, the percentage of gates the visual policy successfully crosses; *Mean Gate Error (MGE)*, average distance between the UAV and the gate center at the time of gate crossing.

Using FalconGym 2.0's Edit API, we design three dynamically feasible observable tracks (Figure 1): (i) *Spatial-S*, requiring consecutive sharp turns with altitude changes; (ii) *Random*, where UAV largely maintains heading while alternating left/right gates and adjusting height; (iii) *Moving*, where one gate translates from right to upper-left at $2\,\mathrm{m/s}$. For each track, we selected 10 slightly different initial poses to evaluate and report the average SR and MGE in Table I.

We implemented five policies and evaluate on their performance in terms of SR, MGE and generalization. First, the state-based expert [21] has access to full state and all gate poses. As expected, it attains the best SR/MGE and generalize well across all tracks. Then we implemented two state-of-the-art visual baselines [3], [5]. [3] trains the visual policy in a NeRF-based simulation, we adopt their architecture and re-trained the visual policy in our FalconGym 2.0 with the help of our Edit API. We also replace their IMU input with the past control, because IMU is not applicable for the Dubins airplane model. [5] uses a Swin-transformer-based gate detector, however, since the code is not publicly available, we replaced the gate detector with our Mask Detector describe in Section III-B. We also modify the reward function by tuning the hyperparameters to make it work in our cases. We faithfully replicate the controller architecture based on their paper and our produced results, e.g. Figure 5, does match their reported behavior. Since both visual baselines acknowledge in their papers that they require training and testing on the same track, to further evaluate their method's performance on unseen tracks, we train those baselines on the Spatial-S track and evaluate on all three tracks. The result shows that while both baselines performs well in the training Spatial-S track, they do not generalize well to unseen tracks Random and Moving, as shown in Table I. We also train and test the baselines again on the exact Random track, and find that they can both achieve 100% success, which confirms that their method are overfitting to training tracks.

TABLE I: Evaluation of five policies based on Success Rate (SR) and Mean Gate Error (MGE) in the UAV Case Study. Success criteria requires distance at gate plane crossing (MGE) $\leq$ 80 cm.

	FalconGym 2.0				
Track	Method	Vision?	Generalize?	SR ↑	MGE (cm) ↓
Spatial-S	State-based	✗	✓	100%	15
	Baseline A [3]	✓	✗	100%	24
	Baseline B [5]	✓	✗	100%	20
	Ours (w/o PGR)	✓	✓	100%	47
	Ours (w/ PGR)	✓	✓	**100%**	**19**
Random	State-based	✗	✓	100%	8
	Baseline A [3]	✓	✗	0%	N/A
	Baseline B [5]	✓	✗	50%	47
	Ours (w/o PGR)	✓	✓	100%	33
	Ours (w/ PGR)	✓	✓	**100%**	**21**
Moving	State-based	✗	✓	100%	14
	Baseline A [3]	✓	✗	0%	N/A
	Baseline B [5]	✓	✗	0%	N/A
	Ours (w/o PGR)	✓	✓	100%	47
	Ours (w/ PGR)	✓	✓	**100%**	**15**

In contrast, our methods, both our uniformly sampling approach and performance-guided refinement approach generalize to unseen tracks with a *single* visual policy. In fact, all three evaluation tracks are unseen to our visual policy, since our training only relies on two-gate track layouts (Section III-B). The "Ours (w/o PGR)" approach refers to our method by uniformly sampling the gate space to generate 300k dynamically feasible and observable two-gate tracks and train the visual policy by imitation. The "Ours (w/ PGR)" approach utilized the performance-guided refinement in Section III-C. Specifically, we partition $G \subset \mathbb{R}^8$ (each gate has (x, y, z, yaw)) into $(4 \times 4 \times 3 \times 3)^2 \approx 20$k grids. In the first iteration we draw 5 samples per grid to form 100k training tracks; we then run PGR for T=3 iterations with β=0.05 in Algorithm 1. Both our variants achieve high SR and generalize well on all tracks, while PGR further reduces MGE, indicating safer crossings and better performance.

Furthermore, we assess each policy's sensitivity to gate position changes using domain randomization enabled by FalconGym 2.0's Edit API, as shown in Figure 5. For fair comparison, we conduct this ablation study on the Spatial-S track on which both baselines are trained, and evaluate

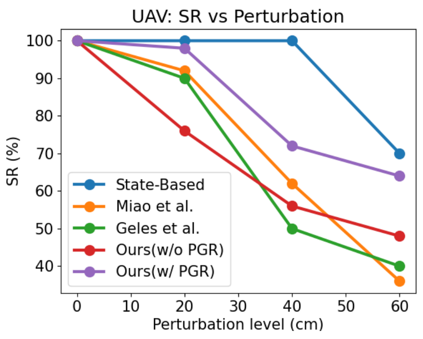

Fig. 5: **Robustness to gate-pose perturbations on *Spatial-S* track in FalconGym 2.0.** For a perturbation level a cm, each gate is independently shifted by a random 3D offset $\delta \in [-a, a]^3$. For each perturbation level, we run all five policies on 10 randomized tracks (50 gates total) and report the Success Rate (SR).

perturbed variants of that track. At a given perturbation level a (cm), every gate is independently shifted by a 3D offset $\delta \in [-a, a]^3$ (applied along x, y, and z direction with random signs). We generate 10 perturbed tracks (50 gates total) for each perturbation level and run all five policies, reporting Success Rate (SR). Note that since we originally designed Spatial-S track to be challenging with sharp turns and altitude changes, increasing a demands sharper turns and pitch changes, so SR decreases for all methods. For high perturbation level, even state-based expert fails because of dynamically infeasible tracks. Figure 5 shows that our PGR approach remains more robust in gate poses perturbation than the visual baselines and uniform sampling. PGR tracks the expert's trend closely with only slight SR gap, which is expected since our visual policy imitates state-based expert.

B. Case Study 2: Quadrotor

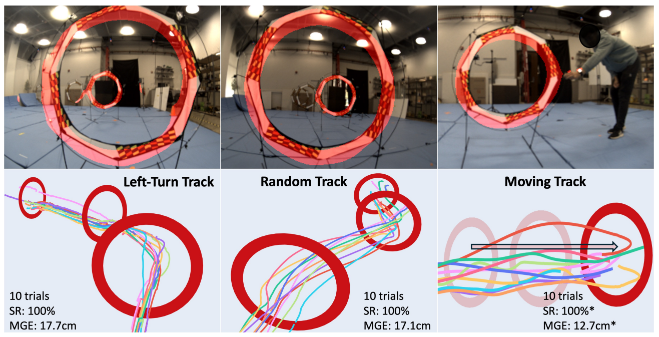

Fig. 6: **Trajectory plots for the quadrotor case study on real hardware.** Visual policy learned in FalconGym2.0 can be zero-shot transferred to real hardware. We disable the forward velocity on Moving track for safety reasons, evaluating lateral tracking only.

1) Performance in FalconGym 2.0: We model the quadrotor with the standard 12-state dynamics of [22]. The state comprises position, linear velocity, attitude, and angular velocity. The state-based expert controller is also from [22], which we set a constant 1m/s forward velocity. Among several control modalities, we choose body-frame linear velocities and yaw rate to match the hardware interface used for zero-shot sim-to-real transfer in Section IV-B.2. Our quadrotor flight arena measures $6 \times 6 \times 3\,\mathrm{m}$, as shown in Figure 6. The circle gates we used have a $78\,\mathrm{cm}$ inner diameter; the quadrotor width is $18\,\mathrm{cm}$. Therefore, a crossing is considered *successful* if, at the instant the quadrotor intersects the gate plane, its distance to the gate center (MGE) $\leq 30cm$. We adapt the same metrics as in the fixed-wing study: *Success Rate (SR)* and *Mean Gate Error (MGE)*.

We again design 3 dynamically feasible and observable tracks (Figure 4) using FalconGym 2.0's Edit API: (i). *Left-Turn*, which involves sustained turning; (ii). *Random*, where quadrotors need to maintain general heading while alternating left/right gates; (iii). *Moving*, where one gate translates to the right at $0.25\,\mathrm{m/s}$.

TABLE II: Evaluation of five policies based on SR and MGE in both FalconGym 2.0 and real world for the quadrotor case study. * indicates lateral-tracking-only for Moving track for safety reasons. Success criteria requires MGE $\leq$ 30 cm.

		FalconGym 2.0		Real World	
Track	Method	SR ↑	MGE (cm) ↓	SR ↑	MGE (cm) ↓
Left-Turn	State-based	100%	5.8	100%	11.6
	Baseline A [3]	100%	14.3	/	/
	Baseline B [5]	100%	17.0	/	/
	Ours (w/o PGR)	100%	11.3	93.3%	21.4
	Ours (w/ PGR)	**100%**	**9.5**	**96.7%**	**17.7**
Random	State-based	100%	5.7	100%	5.9
	Baseline A [3]	67.7%	32.3	/	/
	Baseline B [5]	25%	40.0	/	/
	Ours (w/o PGR)	100%	11.4	96.7%	16.2
	Ours (w/ PGR)	**100%**	**9.5**	**100%**	**17.1**
Moving	State-based	100%	2.0	100%*	7.4*
	Baseline A [3]	0%	N/A	/	/
	Baseline B [5]	0%	N/A	/	/
	Ours (w/o PGR)	100%	24.1	100%*	16.8*
	Ours (w/ PGR)	**100%**	**20.7**	**100%***	**12.7***

Baselines are implemented as in the fixed-wing study but trained with the quadrotor GSplat data, with the modifications described in Section IV-A to fit our pipeline. While our exact visual policy (perception and controller) also works in the quadrotor case study, we slightly modify it by replacing the imitation-learned controller with a classical controller that (i) filters the perception noise and retains only the largest connected component (closest gate) in the predicted mask and (ii) commands the aerial vehicle to follow the centroid of that component. This engineering trade-off is necessary because the hardware quadrotor (Section IV-B.2) cannot execute two neural networks sequentially at the required closed-loop rate. Therefore, we keep the U-Net for perception and replace the second network with a classical controller. As a result, instead of applying PGR to the entire visual policy (perception and controller) as in the fixed-wing UAV case study, we apply it only to the perception module.

As in the fixed-wing UAV experiments, we evaluate all five policies on three tracks with ten different initial conditions. Again, both baselines perform well on the training *Left-Turn* track but overfit and degrade on the unseen *Random* and *Moving* tracks, as shown in Table II. By contrast, both our uniform sampling (Ours w/o PGR) and performance-guided refinement (Ours w/ PGR) maintain 100% SR across all tracks. While PGR further improves MGE, the margin over uniform sampling is smaller than in the fixed-wing case. We believe this is due to: (i) PGR is only applied to the perception module (rather than the full perception-control stack) and (ii) the quadrotor's smaller workspace with coarser grid discretization, i.e., $\left(3 \times 3 \times 2 \times 2\right)^2 = 1{,}296$ grids.

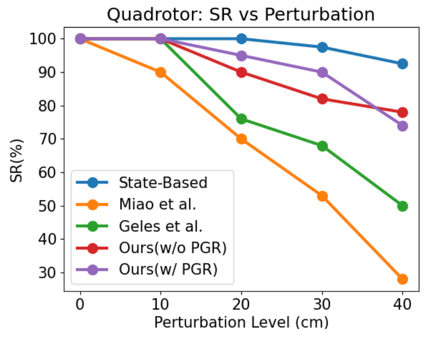

Fig. 7: **Gate perturbations on Left-Turn track in FalconGym 2.0.** For each perturbation level, we run all five policies on 10 randomized tracks (30 gates total) and report the Success Rate (SR).

We also conduct a gate-pose perturbation study similar to the fixed-wing ablation study on the Left-Turn track in FalconGym 2.0. Figure 7 shows that baselines' SR drops rapidly as perturbation level increases, whereas our policies remain substantially more robust and closely track the expert's. The incremental gain from PGR over uniform sampling is modest, again likely due to the partial refinement on the perception only and the smaller space and number of grids.

2) Zero-shot Sim-to-Real Transfer: We zero-shot deployed the trained perception module and same classical controller used in FalconGym 2.0 on a $280\,\mathrm{g}$ ModalAI$^{®}$ Starling 2 quadrotor (upper-right of Figure 2). The quadrotor is equipped with a PX4 flight controller, a VOXL 2$^{®}$ onboard computer with an Adreno 650 GPU for neural inference, and a 12 MP RGB camera. The onboard closed-loop frequency is 8 Hz when running U-Net perception and classical controller.

Although our visual policy does not require the tracks to be the same as in training or in FalconGym 2.0, for a fair sim-to-real comparison, we arranged the hardware courses to mirror the simulation layouts: *Left-Turn*, *Random*, and *Moving*. Due to the small indoor arena and observability assumptions, for *Left-Turn* and *Random* we cross only three gates and reserve the fourth as a navigational waypoint to satisfy the observability requirement. For *Moving* track, a human operator slowly pulls the gate laterally at around $0.25\,\mathrm{m/s}$; we disable forward velocity for safety reasons and evaluate average lateral tracking error of the gate centroid as MGE. We use a motion-capture (mocap) to log the quadrotor's poses and the moving gate's trajectories. While we also use mocap to implement the state-based controller baseline, it is not used in FalconGym 2.0's construction or in any of the vision-based closed-loop. Real-world onboard views and quadrotor trajectories are visualized in Figure 6.

Due to safety concerns, differing control modalities, and limited onboard computation power, we do not run the visual baselines on hardware. We evaluate the remaining three policies with the same settings in FalconGym 2.0: each policy is run on each track with 10 different initial conditions, and then we report the SR and MGE in Table II. Although both our visual-policy variants perform slightly worse than in simulation, they maintain high success rates (SR $\geq 93\%$) across all tracks. The primary failure modes are (i) perception errors where the U-Net confuses a background ladder with a gate and (ii) latency limits from the 8 Hz closed loop. We mark the Moving track results with * to indicate lateral-tracking-only evaluation in hardware (no gate crossing due to safety reasons), which most likely causes lower MGE than the full crossing achieved in FalconGym 2.0. Consistent with simulation and the UAV case study, PGR shows a slight improvement over uniform sampling in SR and MGE. And both our visual policy variants follow the state-based controller performance closely in 3 tracks.

V. CONCLUSIONS

We introduced *FalconGym 2.0* (code to be released upon publication), a GSplat-based photorealistic simulation framework that provides an *Edit API* for programmatic object transforms. Leveraging this editability, we proposed a *performance-guided refinement* (PGR) algorithm that concentrates visual policy's training on challenging tracks and iteratively improves its performance. Across two case studies (fixed-wing UAVs and quadrotors) with different dynamics and environments, our visual policy generalizes to different tracks, outperforms baselines by achieving 100% SR on unseen tracks in FalconGym 2.0 and showing better robustness to gate-pose perturbations. Finally, we demonstrate zero-shot sim-to-real transfer on a hardware quadrotor with 98.6% SR over three tracks (70 gates).

In future work, we plan to: relax the *observability* assumption to handle occluded gates; evaluate higher-speed flights for the quadrotor case study; incorporate realistic UAV dynamics to enable fixed-wing sim-to-real transfer; and distill the perception and controller neural networks to deploy the full two-network stack onboard.

REFERENCES

[1] J. Low, M. Adang, J. Yu, K. Nagami, and M. Schwager, "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.

[2] B. Kerbl, G. Kopanas, T. Leimkühler, and G. Drettakis, "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/

[3] Y. Miao, W. Shen, and S. Mitra, "Falcongym: A photorealistic simulation framework for zero-shot sim-to-real vision-based quadrotor navigation," in *IEEE/RSJ International Conference on Intelligent Robots and Systems*, Hangzhou, China, October 2025.

[4] B. Mildenhall, P. P. Srinivasan, M. Tancik, J. T. Barron, R. Ramamoorthi, and R. Ng, "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

[5] I. Geles, L. Bauersfeld, A. Romero, J. Xing, and D. Scaramuzza, "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, and E. Coronado, Eds., 2024. [Online]. Available: https://doi.org/10.15607/RSS.2024.XX.082

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

[7] P. Soviany, R. T. Ionescu, P. Rota, and N. Sebe, "Curriculum learning: A survey," 2022. [Online]. Available: https://arxiv.org/abs/2101.10382

[8] A. Byravan, J. Humplik, L. Hasenclever, A. Brussee, F. Nori, T. Haarnoja, B. Moran, S. Bohez, F. Sadeghi, B. Vujatovic, and N. M. O. Heess, "Nerf2real: Sim2real transfer of vision-guided bipedal motion skills using neural radiance fields," *2023 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 9362–9369, 2022. [Online]. Available: https://api.semanticscholar.org/CorpusID:252815541

[9] M. Torne, A. Simeonov, Z. Li, A. Chan, T. Chen, A. Gupta, and P. Agrawal, "Reconciling reality through simulation: A real-to-sim-to-real approach for robust manipulation," *Arxiv*, 2024.

[10] Z. Xie, Z. Liu, Z. Peng, W. Wu, and B. Zhou, "Vid2sim: Realistic and interactive simulation from video for urban navigation," *Preprint*, 2024.

[11] S. Yang, W. Yu, J. Zeng, J. Lv, K. Ren, C. Lu, D. Lin, and J. Pang, "Novel demonstration generation with gaussian splatting enables robust one-shot manipulation," *arXiv preprint arXiv:2504.13175*, 2025.

[12] O. Shorinwa, J. Tucker, A. Smith, A. Swann, T. Chen, R. Firoozi, M. D. Kennedy, and M. Schwager, "Splat-mover: Multi-stage, open-vocabulary robotic manipulation via editable gaussian splatting," 2024.

[13] R. Portelas, C. Colas, K. Hofmann, and P.-Y. Oudeyer, "Teacher algorithms for curriculum learning of deep rl in continuously parameterized environments," in *Proceedings of the Conference on Robot Learning*, ser. Proceedings of Machine Learning Research, L. P. Kaelbling, D. Kragic, and K. Sugiura, Eds., vol. 100. PMLR, 30 Oct–01 Nov 2020, pp. 835–853. [Online]. Available: https://proceedings.mlr.press/v100/portelas20a.html

[14] H. Wang, J. Xing, N. Messikommer, and D. Scaramuzza, "Environment as policy: Learning to race in unseen tracks," 2025. [Online]. Available: https://arxiv.org/abs/2410.22308

[15] Y. Chen, Z. Chen, C. Zhang, F. Wang, X. Yang, Y. Wang, Z. Cai, L. Yang, H. Liu, and G. Lin, "Gaussianeditor: Swift and controllable 3d editing with gaussian splatting," 2023. [Online]. Available: https://arxiv.org/abs/2311.14521

[16] Y. Wang, X. Yi, Z. Wu, N. Zhao, L. Chen, and H. Zhang, "View-consistent 3d editing with gaussian splatting," 2025. [Online]. Available: https://arxiv.org/abs/2403.11868

[17] C. Vachha and A. Haque, "Instruct-gs2gs: Editing 3d gaussian splats with instructions," 2024. [Online]. Available: https://instruct-gs2gs.github.io/

[18] J. L. Schönberger and J.-M. Frahm, "Structure-from-motion revisited," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 4104–4113.

[19] W. Kabsch, "A discussion of the solution for the best rotation to relate two sets of vectors," *Acta Crystallographica Section A*, vol. 34, no. 5, pp. 827–828, Sept. 1976. [Online]. Available: http://dx.doi.org/10.1107/S0567739478001680

[20] M. Tancik, E. Weber, E. Ng, R. Li, B. Yi, J. Kerr, T. Wang, A. Kristoffersen, J. Austin, K. Salahi, A. Ahuja, D. McAllister, and A. Kanazawa, "Nerfstudio: A modular framework for neural radiance field development," in *ACM SIGGRAPH 2023 Conference Proceedings*, ser. SIGGRAPH '23, 2023.

[21] M. Owen, R. W. Beard, and T. W. McLain, *Implementing Dubins Airplane Paths on Fixed-Wing UAVs*. Dordrecht: Springer Netherlands, 2015, pp. 1677–1701.

[22] F. Sabatino, "Quadrotor control: modeling, nonlinear control design, and simulation," 2015. [Online]. Available: https://api.semanticscholar.org/CorpusID:61413561

[23] 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*, N. Navab, J. Hornegger, W. M. Wells, and A. F. Frangi, Eds. Springer International Publishing, 2015, pp. 234–241.

[24] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial networks," *Commun. ACM*, vol. 63, no. 11, p. 139–144, Oct. 2020. [Online]. Available: https://doi.org/10.1145/3422622

[25] K. Shimizu, Y. Ishizuka, and J. F. Bard, *Min-Max Problem*. Boston, MA: Springer US, 1997, pp. 271–279.