AirSim Drone Racing Lab

Ratnesh Madaan[1]   RATNESH.MADAAN@MICROSOFT.COM
Nicholas Gyde[1]   V-NIGYDE@MICROSOFT.COM
Sai Vemprala[1]   SAI.VEMPRALA@MICROSOFT.COM
Matthew Brown[1]   V-MATTBR@MICROSOFT.COM
Keiko Nagami[2]   KNAGAMI@STANFORD.EDU
Tim Taubner[2,3]   TAUBNERT@INF.ETHZ.CH
Eric Cristofalo[2]   ECRISTOF@STANFORD.EDU
Davide Scaramuzza[3]   SDAVIDE@IFI.UZH.CH
Mac Schwager[2]   SCHWAGER@STANFORD.EDU
Ashish Kapoor[1]   AKAPOOR@MICROSOFT.COM

[1] *Microsoft, One Microsoft Way, Redmond, WA 98052*

[2] *Multi-Robot Systems Lab, Department of Aeronautics and Astronautics, Stanford University Durand Building, 496 Lomita Mall, Stanford, CA 94305*

[3] *Robotics and Perception Group, Depts. of Informatics and Neuroinformatics, University of Zurich and ETH, Andreasstrasse 15, 8050, Zurich, Switzerland*

Editors: Hugo Jair Escalante and Raia Hadsell

Abstract

Autonomous drone racing is a challenging research problem at the intersection of computer vision, planning, state estimation, and control. We introduce AirSim Drone Racing Lab, a simulation framework for enabling fast prototyping of algorithms for autonomy and enabling machine learning research in this domain, with the goal of reducing the time, money, and risks associated with field robotics. Our framework enables generation of racing tracks in multiple photo-realistic environments, orchestration of drone races, comes with a suite of gate assets, allows for multiple sensor modalities (monocular, depth, neuromorphic events, optical flow), different camera models, and benchmarking of planning, control, computer vision, and learning-based algorithms. We used our framework to host a simulation based drone racing competition at NeurIPS 2019. The competition binaries are available at our github repository[1].

Keywords: Drone Racing, UAV, Robotics, Planning, Perception, Machine Learning

1. Introduction

Machine Learning (ML) methods are increasingly showing promise in addressing challenges in robotics and autonomous systems. Perception-action loops are at the core of these devices, and recent advances in reinforcement learning (RL) and imitation learning (IL) are potentially applicable in this domain. However, these results are often limited to constrained laboratory settings (Hwangbo et al., 2017; Kahn et al., 2017) or focus on simple computer games (Mnih et al., 2013; Silver et al., 2016). ML has the potential to positively accelerate

---

1. https://github.com/microsoft/AirSim-NeurIPS2019-Drone-Racing

Figure 1: Snapshots from a few of our racing environments and tracks.

the field of robotics; however, the barrier to entry in this research area is high due to the financial cost, complexity and risk associated with having robots operating in the real world. In this paper we address the challenge of lowering the barrier to entry for ML researchers into the task of autonomous drone racing (Moon et al., 2019). Our core hypothesis is that simulation can help mitigate the complexity associated with experimenting with flying robots in the real world, and that it is possible to design APIs at an abstraction level targeted to ML researchers.

Started by passionate hobbyists, first person view (FPV) drone racing has gained in popularity over the past few years and is now recognized as a professional sport. Humans are able to fly drones (quadcopters in this context) aggressively through complex and cluttered environments, which may consist of varying number of gates, gate poses, sizes, texture and shapes, using only low resolution, noisy first-person view images while competing against and avoiding other racing drones in flight. The control inputs for the racing drones are angular velocity rates and thrust commands, which require the pilot to constantly stabilize them, unlike velocity or position input modes found on their commercial counterparts.

The UAV research community has recognized drone racing as one of the next important challenges to address because it requires multiple autonomy modules (state estimation, gate detection and pose estimation, trajectory planning, control, and reactive avoidance strategies) to work in harmony with each other. However, quick prototyping of the autonomy stack and verifying its generalization abilities on varied race environments is still a challenge. We develop a simulation framework, AirSim Drone Racing Lab, to address the above. In addition, we used our framework to organize a simulation based drone racing competition at the NeurIPS 2019 Conference, *Game of Drones (GoD)*, with the goal of attracting the ML community to focus on robotic sub-problems of trajectory planning, control, computer vision, and head-to-head competition with an opponent racer.

2. Related Work

The field of robotics has a rich history of challenges and competitions being drivers of research. In recent years, drone-specific programs such as DARPA FLA (2017), MBZIRC Challenge (2016-present), and IMAV Challenge (2009-present) have come into focus, where the objective has generally been to build UAV platforms to address tasks such as navigation, mapping, and pick-and-place autonomously.

Autonomous drone racing has gained substantial momentum over the last few years as a research problem, with multiple approaches being presented to address perception, planning and control to achieve agile, accurate flight (Jung et al., 2018; Kaufmann et al., 2018,
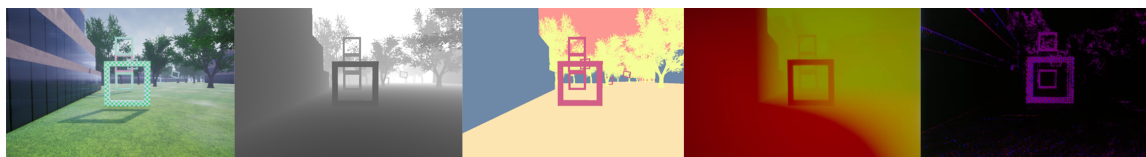
Figure 2: An example showcasing different sensor modalities our framework provides - from left to right: monocular RGB, depth, segmentation, optical flow, and events.

2019; Lin et al., 2019; Loquercio et al., 2019). Research oriented drone racing competitions have also been introduced, the largest among them being the recent AlphaPilot Challenge (2019), which used the MIT FlightGoggles simulation framework (Guerra et al., 2019) in its qualification round and required implementation on their hardware in the later stages. Similarly, an annual hardware oriented drone racing challenge (Moon et al., 2019) has been hosted at the IROS robotics conference since 2016. In addition, multiple datasets (UZH-FPV (Delmerico et al., 2019), Blackbird (Antonini et al., 2018)) have also been released to aid the development of vision based drone racing algorithms.

The goal of our AirSim Drone Racing Lab simulation framework is to help bridge the gap between simulation and reality by utilizing high fidelity graphics and physics simulation. We achieve this by building on top of AirSim (Shah et al., 2018), which leverages Unreal Engine for graphics. The graphical capabilities of Unreal Engine enable generation of photo-realistic environments, benefiting the development of perception algorithms and sim-to-real transfer techniques. For example, Loquercio et al. (2019) attempt sim-to-real transfer by randomizing the visual properties of the underlying Gazebo simulator (Koenig and Howard, 2004) and the RotorS extension (Furrer et al., 2016). However, Gazebo does not quite match the photorealistic capabilities of Unreal Engine.

Through the *Game of Drones (GoD)* competition at NeurIPS 2019, we put the framework to test. While other drone racing competitions prioritized the design of/implementation on hardware racing drones, we believe simulation helps reach wider audiences and connect the machine learning and robotics communities, while allowing participants to focus on autonomy-enabling algorithms. Unlike the AlphaPilot and IROS racing challenges which primarily use a time trial format, the participants in *GoD* raced against a simulated opponent, emphasizing head-to-head competition and bringing the problem statement closer to human FPV drone racing.

3. AirSim Drone Racing Lab

One of the main goals of our framework is to make drone racing accessible to ML researchers and engineers, who have the relevant knowledge of algorithms and software but might not have exposure to the hardware and systems aspects of robotics. To achieve this, we build on AirSim (Shah et al., 2018), a high-fidelity simulation framework for multirotors which implements a lightweight physics engine, flight controller, and inertial sensors; and also comes with photo-realistic camera and depth sensors via Unreal Engine (UE).

While it provides us with a good starting point, drone racing research with AirSim is not possible out of the box. Drone racing is a research problem which involves multiple moving

Figure 3: Sample features from our framework: Left - Environment ground truth via voxel grid, Right - Signed Distance Field gradient visualization for a gate, Middle - Examples of Domain Randomization.

components. Apart from the core AirSim components, one needs a framework to orchestrate drone races, and a set of APIs and features geared towards ML research.

Drone-racing specific needs include multiple *environments* in which tracks can be set up, a library of 3D assets corresponding to *drone gates*, a framework to start and reset *races*, and track the *race progress* (every *gate* has to detect which drone *racer* passed through it), displaying and accessing *race progress* from APIs to monitor score and performance, and an ability to enforce *race rules* such as time *penalties* associated with environment collisions and *disqualifications* in the event of drone-drone collision.

ML specific needs include data acquisition from multiple sensor modalities, generalization of learnt models (whether perception or control oriented); representation, imitation, and reinforcement learning. Specifically, we want our APIs to allow for generation of diverse gate datasets (spawning and destructing gate assets); domain randomization (changing the shape, scale, pose, and texture of gates), and cater for multiple components of autonomy used in drone racing targeted at different communities. To emphasize the last point, we provide high level planning and control APIs for researchers who are interested in solving perception tasks (detecting gates from images and estimating their pose), low-level controls for users interested in learning sensorimotor policies, and support for imitation learning dataset generation. In order to train policies robust to different camera models, we also provide APIs to set camera model parameters - intrinsics, distortion coefficients, and rolling shutter; while also generating ground truth optical flow images.

In addition, high speed drone racing motivates investigation of event cameras due to their high frame rate and dynamic range, asynchronous nature, and lack of motion blur (Rebecq et al., 2018; Gallego et al., 2019). We leverage Unreal Engine's graphics capabilities to draw events at a high rate, and we aim to provide event data in a future release of our full framework[2] (Appendix C). Finally, we will also provide ground truth representations of the full simulated world or a region of interest thereof, in terms of voxel grids, specifically sparse voxel octrees (Laine and Karras, 2010). These voxel grids are converted into signed distance fields (Oleynikova et al., 2017; Han et al., 2019), that provide the signed distance to the

---

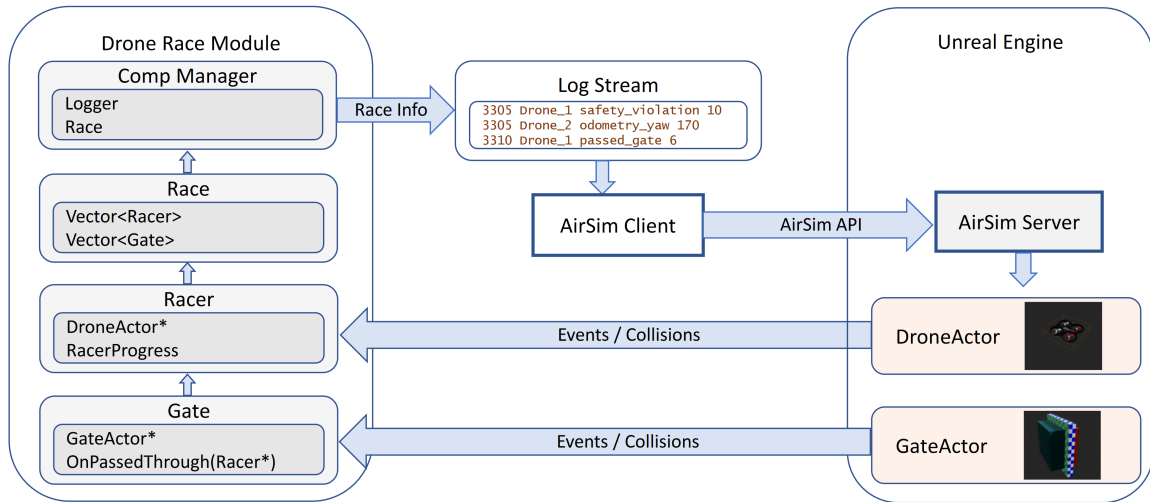2. https://github.com/microsoft/AirSim-Drone-Racing-Lab

Figure 4: AirSim Drone Racing Lab software architecture.

nearest obstacle and the gradient thereof at query locations, which can then be consumed by trajectory optimization based planners.

Software Architecture: A schematic of our software architecure is visualized in Fig. 4. Each race environment contains custom UE actors (`ADrone`, `AGate`) which detect race-related simulation events - drone collisions, crashes, near misses, and gate passthroughs. The actors then relay these events to a standalone C++ race module separated from AirSim, which aids in orchestration of a races. The race module is composed of multiple classes (`Race`, `Racer`, `Gate`), each of which have relevant UE actors (`ADrone`, `AGate`). Each `Racer` owns a `RacerProgress` object, which tracks its competition metrics based on events (for example, a collision event may be interpreted as a disqualification). Finally, the `CompManager` collects the overall race state from all the `RacerProgress` objects, and streams relevant telemetry to a log file. The file is flushed in real time and exists as a channel to inform the AirSim client of the race state in real-time, and serves as the final component in a continuous feedback loop between API control and `Race` state tracking. We note here that for *GoD*, the participants submitted the generated log files to our server where we ran the evaluation scripts, and then updated the leaderboard.

In addition, we allow for changing of UE environments at runtime by cooking them into DLC packages (`.pak` files) and exposing an API. A full list of our APIs is available in Appendix A and our website.

4. Race Tracks and Baselines

Our current release binaries include three Unreal Engine environments - `Soccer Field`, `ZhangJiaJie`, `MSR Building 99` - over which racing tracks are designed, with a cumulative of nine tracks. We used three tracks each in different stages (training, qualification, final round) of *GoD*. In order to benchmark autonomy algorithms, it is important to characterize the complexity of a racing track quantitatively. We introduce two measures of complexity:

Track Name	Curvature Metric (m^{-1})	$v_{max}(m/s)$, $a_{max}(m/s^2)$	moveOnSpline Lap Time (s)	Perception Error (m)
T_Soccer_Field_Easy	0.05	30, 15	58.019	1.673
T_ZhangJiaJie_Medium	0.04	30, 15	125.775	2.097
T_Building_99_Hard	0.09	5, 2	80.699	1.539
Q_Tier_1	0.04	30, 15	110.867	2.845
Q_Tier_2	0.03	30, 15	80.553	1.640
Q_Tier_3	0.02	30, 15	143.830	2.481
F_Tier_1	0.05	30, 15	106.637	2.856
F_Tier_2	0.03	30, 15	113.788	3.774
F_Tier_3	0.02	30, 15	124.135	1.739

Table 1: Track complexity and baseline performance.

curvature per unit length (for planning and control tasks) and next gate visibility (for perception oriented tasks), and then outline baseline algorithms for the same.

Quantifying complexity of a race track: For *planning and control tasks*, we use the curvature measure of a 3D curve and normalize it per unit length. A racing track can be defined by the number of gates, their poses, and the direction vector along which a racer should pass through the gate. We fit a 3rd order spline through the center of all gates. For this three dimensional curve, given by $\gamma(t) = (x(t), y(t), z(t))$, we calculate the instantaneous curvature as

$$\kappa = \frac{\sqrt{(z''y' - y''z')^2 + (x''z' - z''x')^2 + (y''x' - x''y')^2}}{(x'^2 + y'^2 + z'^2)^{\frac{3}{2}}}$$

We plot the instantaneous curvature against track length (Fig. 6a), and define a scalar track *Curvature Metric* quantifying its cumulative complexity – AUC (Area Under the Curve) *normalized* by the track length, and report it in Table 1. For *perception tasks*, we define hardness by *Gate Visibility* – the number of pixels belonging to the next gate in the RGB image divided by the total pixel area of the image (Fig. 6b). This gives an approximate measure of the information available of the next gate in the FPV image when flying along the baseline trajectory. We note that this is only an indicative metric and actual gate visibility can vary with camera intrinsic parameters such as field of view. We refer the reader to Appendix B for more details.

Trajectory planning and tracking baseline: Researchers focusing on perception oriented tasks would not want to spend a lot of their resources on planning and control oriented tasks. In addition, it is important that the training datasets they use are a representative sample of test time, when a drone is flying through gates. To address these needs, we introduce a high level API moveOnSpline, which consumes a list of 3D waypoints along with desired velocity and acceleration limits, and does minimum jerk trajectory planning and pure pursuit tracking control in the backend.

Exploiting quadrotors' differential flatness property (Mellinger and Kumar (2011)), we plan in $\mathbb{R}^3$ for minimum jerk trajectories which are splines composed of piece-wise polynomials with C^2 continuity. We plan for yaw angles separately and assign them to follow the tangents of the fitted spline. We refer the reader to (Oleynikova et al., 2016; Burri et al., 2015; Richter et al., 2016) for details. To track the trajectory, we implement a pure-pursuit controller, which minimizes position and velocity errors in the cross-track, along-track, and world z dimensions. We report the lap times in Table 1, along with the maximum specified

velocity and acceleration for each track (for the *Building 99* track, these values are much lower due to tight corridors).

In the context of drone racing, it is useful to specify a user-defined velocity direction vector while passing a user-defined waypoint (for example – flying along the gate normal direction through a gate center). For this use case, we expose a `moveOnSplineVelConstraints` API which takes in velocity vector constraints in addition to each 3D waypoint.

Gate detection baseline: We use a recursive filtering algorithm to continuously estimate the relative pose of the next gate with respect to the drone. The gates used in the competition tracks are rectangular, planar, and have similar color and texture. Under the assumptions of known gate dimensions, we use planar homography to estimate the center of the gate (Hartley and Zisserman (2004)). We capture a baseline image of a gate under the assumption of known gate dimensions and relative distance from the camera, with the camera's optical axis normal to the gate cross section. With this setup, the center pixel of the baseline image can be mapped to the gate center.

During flight, we use color thresholding to extract a gate mask and retrieve the best estimates of the four gate corners. The homography matrix is determined to map the center gate pixel of the baseline image to that of a different image, using the four gate corner pixel points as a reference. From here, a point to point correspondence is used to find the 3D coordinate of the gate center. The measurements of the next gate's center location are fed into a Kalman Filter to obtain improved estimates.

In order to quantify the performance of this baseline, we collect 1000 measurements of gate centers, by flying the quadrotor through each track using `moveOnSpline` (assuming ground truth of gate positions), and report the mean Euclidean distance between gate center measurements and ground truth in Table 1.

5. Game of Drones Competition at NeurIPS 2019

We used our framework to host a simulation based drone racing competition at NeurIPS 2019. The competition was scoped to focus on three core areas pertinent to autonomous drone racing - perception, trajectory planning and control, and head-tp-head competition with a *single* competitor drone. (We note that our framework allows for an arbitrary number of drones in a single race, but *GoD* was limited to two racers.) In each tier, the objective is to pass through all the gates in minimum possible time, without any collision with the environment. If an opponent racer is present in the tier, there is an additional objective to not collide with it. In the case of collision with the gates or any other objects in the environment, a collision penalty to the total lap time is applied. In the case of drone-drone collision, the trailing drone is disqualified. The participants are first ranked according to the number of gates flown through, followed by lap time. The competition specific binaries and baselines' implementations are available at our github repository[3]. We now explain each tier, and a competition specific game-theoretic planning baseline.

Tier 1 – Planning only: The participant's drone races tête-à-tête with an opponent racer. Ground truth for state estimation and environment is provided via our APIs, in the form of the odometry (position and velocity) of the participant and the opponent drones, and the poses of all the gates. The opponent racer follows a minimum jerk trajectory via

---

3. https://github.com/microsoft/AirSim-NeurIPS2019-Drone-Racing

(a) Tier 1   (b) Tier 2 and 3

Figure 5: In Tier 1, ground truth of gate poses is given. In Tiers 2 and 3, the poses are corrupted with noise.

`moveOnSpline`, and goes through randomized waypoints selected in each gate's cross section. Hence, the opponent's trajectory varies at every run.

Tier 2 – Perception only: In this tier, the gate poses returned by the API are corrupted with noise as shown in Fig. 5b, and there is no opponent drone. The ground truth state estimate of the participant drone is still available. The next gate is not always in view, but the noisy pose returned by our API help steer the participants roughly in the right direction, after which vision-based control would be necessary.

Tier 3 – Perception and Planning: This tier has both noisy gate poses and an opponent racer, and is essentially a combination of Tier 1 and 2.

Game theoretic planning baseline: Head-to-head drone racing in Tiers 1 and 3 of *GoD* brings with it an inherent competitor interaction which can be addressed via game theory. We implemented a Game Theoretic Planner baseline based on (Spica et al. (2018)) which can solve the two-player drone racing problem. We refer the reader to our github repository for the implementation of this baseline.

Organizational details: We ran *GoD* in two stages: a qualification and a final round. Initially, a set of *training* binaries with our three UE environments was released to the participants, for prototyping on user-defined race tracks and user-defined race tiers. Then, we released a new set of binaries for the *qualification* stage with one race track for each tiers, where participants were asked to make submissions for their choice of tiers. We saw 117 teams registering for the competition worldwide, with 16 unique entries that showed up on the *qualification* leaderboard. Lastly, the *final* round was conducted with another set of racetracks, again with one track for each tier.

Winning teams' approaches: The detailed reports, talks of the winning teams, and the leaderboards can be found at our website[4]. We briefly summarize their approaches here[5].

Tier 1 winner (*Team Dedalé*) won by a significant margin because of two key components in their approach - (1) planning a globally optimal trajectory by allowing it to pass anywhere through the gate cross-sections (as opposed to forcing an artificial position constraint at the

---

4. https://microsoft.github.io/AirSim-NeurIPS2019-Drone-Racing

5. We do note that this section is not our work, and the relevant publications by the winning teams should be cited if the reader wishes to build upon their work. However, we believe it is important to disseminate the core components of their approaches.

gate center), (2) modeling of drag forces, (3) a non-linear model predictive controller (Kamel et al., 2017). It is a known fact that the drag forces acting on any body (including a quadrotor) become significant at high speeds (Spedicato and Notarstefano, 2017), and that the actual velocity of the quadrotor does not match the reference velocity generated by methods which do not account for drag (Richter et al., 2016; Burri et al., 2015).

Tiers 2 and 3 winner (*Team Sangyun*) also won by a significant margin. For gate detection, they use MobileNet-SSD (Howard et al., 2017; Liu et al., 2016) trained on a a synthetic dataset generated from our training binaries. The predicted gate bounding box is compressed into a vectorized state and fed into a policy network, which is trained in an actor-critic fashion with the reward function pushing the policy to fly through the center of the next gate. They had a head start on their RL policy by using pre-trained weights from their previous work on AirSim (Shin et al., 2020). When flying between gates, they use genetic algorithms (Man et al., 1996) for optimizing the inputs parameters to our `moveOnSpline` API, and suggested that this helped a lot in improving their lap times. Between gates, they fly to a weighted sum of the current position of the drone and the given noisy gate pose.

Tier 2 and 3 second position (*Team USRG*) also used MobileNet-SSD for the perception problem, but in addition to detecting the closest gate, they also detect competitor drone. For control, they used a hybrid scheme which switches between position based control (when there is no gate detected) and a velocity based control (when next gate is visible). For position based control, they use the weighted sum strategy similar to *Team Sangyun*, whereas the velocity based control is essentially a rule-based servoing method to fly through the center of the next gate (Jung et al., 2018).

6. Conclusion

We introduce a simulation framework targeted at the domain of autonomous drone racing, with the goal of reducing the entry barrier for the machine learning community and benchmarking of various modules of autonomy in a simulation environment. We build on Microsoft AirSim, and extend its capabilities by adding the ability to orchestrate drone races, exposing additional input modalities (e.g. event cameras and optical flow), customizing camera sensor models, providing environment ground truth, marker visualizations, high level trajectory planning and tracking features, and modifying flight controller gains. We used our framework to host a simulation-only drone racing competition at NeurIPS 2019, which saw a wide range of participation and corresponding approaches.

References

AlphaPilot Challenge. 2019. https://www.herox.com/alphapilot.

Amado Antonini, Winter Guerra, Varun Murali, Thomas Sayre-McCord, and Sertac Karaman. The blackbird dataset: A large-scale dataset for uav perception in aggressive flight. In *2018 International Symposium on Experimental Robotics (ISER)*, 2018.

Michael Burri, Helen Oleynikova, , Markus W. Achtelik, and Roland Siegwart. Real-time visual-inertial mapping, re-localization and planning onboard mavs in unknown environ-

ments. In *Intelligent Robots and Systems (IROS 2015), 2015 IEEE/RSJ International Conference on*, Sept 2015.

DARPA FLA. 2017. https://www.darpa.mil/program/fast-lightweight-autonomy.

Jeffrey Delmerico, Titus Cieslewski, Henri Rebecq, Matthias Faessler, and Davide Scaramuzza. Are we ready for autonomous drone racing? the uzhfpv drone racing dataset. In *IEEE Int. Conf. Robot. Autom.(ICRA)*, 2019.

Philipp Fischer, Alexey Dosovitskiy, Eddy Ilg, Philip Häusser, Caner Hazırbaş, Vladimir Golkov, Patrick Van der Smagt, Daniel Cremers, and Thomas Brox. Flownet: Learning optical flow with convolutional networks. *arXiv preprint arXiv:1504.06852*, 2015.

Fadri Furrer, Michael Burri, Markus Achtelik, and Roland Siegwart. Rotors—a modular gazebo mav simulator framework. In *Robot Operating System (ROS)*, pages 595–625. Springer, 2016.

Guillermo Gallego, Tobi Delbruck, Garrick Orchard, Chiara Bartolozzi, Brian Taba, Andrea Censi, Stefan Leutenegger, Andrew Davison, Joerg Conradt, Kostas Daniilidis, et al. Event-based vision: A survey. *arXiv preprint arXiv:1904.08405*, 2019.

Winter Guerra, Ezra Tal, Varun Murali, Gilhyun Ryou, and Sertac Karaman. Flightgoggles: Photorealistic sensor simulation for perception-driven robotics using photogrammetry and virtual reality. *arXiv preprint arXiv:1905.11377*, 2019.

Luxin Han, Fei Gao, Boyu Zhou, and Shaojie Shen. Fiesta: Fast incremental euclidean distance fields for online motion planning of aerial robots. *arXiv preprint arXiv:1903.02144*, 2019.

R. I. Hartley and A. Zisserman. *Multiple View Geometry in Computer Vision*. Cambridge University Press, ISBN: 0521540518, second edition, 2004.

Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.

Jemin Hwangbo, Inkyu Sa, Roland Siegwart, and Marco Hutter. Control of a quadrotor with reinforcement learning. *IEEE Robotics and Automation Letters*, 2(4):2096–2103, 2017.

IMAV Challenge. 2009-present. http://www.imavs.org.

Sunggoo Jung, Sunyou Hwang, Heemin Shin, and David Hyunchul Shim. Perception, guidance, and navigation for indoor autonomous drone racing using deep learning. *IEEE Robotics and Automation Letters*, 3(3):2539–2544, 2018.

Gregory Kahn, Adam Villaflor, Vitchyr Pong, Pieter Abbeel, and Sergey Levine. Uncertainty-aware reinforcement learning for collision avoidance. *arXiv preprint arXiv:1702.01182*, 2017.

Mina Kamel, Michael Burri, and Roland Siegwart. Linear vs nonlinear mpc for trajectory tracking applied to rotary wing micro aerial vehicles. *IFAC-PapersOnLine*, 50(1):3463–3469, 2017.

Elia Kaufmann, Antonio Loquercio, Rene Ranftl, Alexey Dosovitskiy, Vladlen Koltun, and Davide Scaramuzza. Deep drone racing: Learning agile flight in dynamic environments. In *Conference on Robot Learning*, pages 133–145, 2018.

Elia Kaufmann, Mathias Gehrig, Philipp Foehn, René Ranftl, Alexey Dosovitskiy, Vladlen Koltun, and Davide Scaramuzza. Beauty and the beast: Optimal methods meet learning for drone racing. In *2019 International Conference on Robotics and Automation (ICRA)*, pages 690–696. IEEE, 2019.

Nathan Koenig and Andrew 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)*, volume 3, pages 2149–2154. IEEE, 2004.

Samuli Laine and Tero Karras. Efficient sparse voxel octrees. *IEEE Transactions on Visualization and Computer Graphics*, 17(8):1048–1059, 2010.

Jiarong Lin, Luqi Wang, Fei Gao, Shaojie Shen, and Fu Zhang. Flying through a narrow gap using neural network: an end-to-end planning and control approach. *arXiv preprint arXiv:1903.09088*, 2019.

Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C Berg. Ssd: Single shot multibox detector. In *European conference on computer vision*, pages 21–37. Springer, 2016.

Antonio Loquercio, Elia Kaufmann, René Ranftl, Alexey Dosovitskiy, Vladlen Koltun, and Davide Scaramuzza. Deep drone racing: From simulation to reality with domain randomization. *IEEE Transactions on Robotics*, 2019.

Kim-Fung Man, Kit-Sang Tang, and Sam Kwong. Genetic algorithms: concepts and applications [in engineering design]. *IEEE transactions on Industrial Electronics*, 43(5):519–534, 1996.

Daniel Maurer and Andrés Bruhn. Proflow: Learning to predict optical flow. *arXiv preprint arXiv:1806.00800*, 2018.

MBZIRC Challenge. 2016-present. http://mbzirc.com.

Daniel Mellinger and Vijay Kumar. Minimum snap trajectory generation and control for quadrotors. In *2011 IEEE International Conference on Robotics and Automation*, pages 2520–2525. IEEE, 2011.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

Hyungpil Moon, Jose Martinez-Carranza, Titus Cieslewski, Matthias Faessler, Davide Falanga, Alessandro Simovic, Davide Scaramuzza, Shuo Li, Michael Ozo, Christophe De Wagter, et al. Challenges and implemented technologies used in autonomous drone racing. *Intelligent Service Robotics*, 12(2):137–148, 2019.

Helen Oleynikova, Michael Burri, Zachary Taylor, Juan Nieto, Roland Siegwart, and Enric Galceran. Continuous-time trajectory optimization for online uav replanning. In *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 5332–5339. IEEE, 2016.

Helen Oleynikova, Zachary Taylor, Marius Fehr, Roland Siegwart, and Juan Nieto. Voxblox: Incremental 3d euclidean signed distance fields for on-board mav planning. In *2017 Ieee/rsj International Conference on Intelligent Robots and Systems (iros)*, pages 1366–1373. IEEE, 2017.

Henri Rebecq, Daniel Gehrig, and Davide Scaramuzza. ESIM: an open event camera simulator. *Conf. on Robotics Learning (CoRL)*, October 2018.

Charles Richter, Adam Bry, and Nicholas Roy. Polynomial trajectory planning for aggressive quadrotor flight in dense indoor environments. In *Robotics Research*, pages 649–666. Springer, 2016.

Shital Shah, Debadeepta Dey, Chris Lovett, and Ashish Kapoor. Airsim: High-fidelity visual and physical simulation for autonomous vehicles. In *Field and service robotics*, pages 621–635. Springer, 2018.

Sang-Yun Shin, Yong-Won Kang, and Yong-Guk Kim. Reward-driven u-net training for obstacle avoidance drone. *Expert Systems with Applications*, 143:113064, 2020.

David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484, 2016.

Sara Spedicato and Giuseppe Notarstefano. Minimum-time trajectory generation for quadrotors in constrained environments. *IEEE Transactions on Control Systems Technology*, 26(4):1335–1344, 2017.

Riccardo Spica, Davide Falanga, Eric Cristofalo, Eduardo Montijano, Davide Scaramuzza, and Mac Schwager. A real-time game theoretic planner for autonomous two-player drone racing. In *Robotics: Science and Systems (RSS)*, 2018.

Appendix A. AirSim Drone Racing API

We highlight the main components of AirSim Drone Racing Lab API below, which augment the existing AirSim API functions. Note: APIs marked with a '#' superscript are not available in the GoD binaries, but we will release them along with our full framework at https://github.com/microsoft/AirSim-Drone-Racing-Lab:

- Racer API:
  - Low level control:
    * Angle rate setpoints: `moveByAngleRatesThrottle, moveByAngleRatesZ`
    * Angle setpoints: `moveByRollPitchYawThrottle, moveByRollPitchYawZ`
    * Roll, pitch angles and yaw rate: `moveByRollPitchYawrateThrottle, moveByRollPitchYawrateZ`
    * Set motor RPM#: `setMotorSpeeds`
    * Set low level controller gains: `setAngleRateControllerGains, setAngleLevelControllerGains`
  - Medium level control:
    * Velocity setpoints: `moveByVelocity, moveByVelocityZ`
    * Position setpoints: `moveToPosition, moveToZ, moveOnPath`
    * Set medium level control gains: `setVelocityControllerGains, setPositionControllerGains`
  - High level - trajectory planning and tracking:
    * Fit a spline through waypoints: `moveOnSpline, moveOnSplineVelConstraints`
    * Track a trajectory: `trackTrajectory`
    * Set control gains for tracking: `setTrajectoryTrackerGains`
- Environment API:
  - Race environment API:
    * Load map: `simLoadlevel`
    * Spawning and destruction of gates: `simSpawnObject, simDestroyObject`
    * Gate size ground truth: `simGetNominalGateInner/OuterDimensions`
  - Object API: (To facilitate dataset generation)
    * Object pose and scale: `simSet/GetObjectPose, simSet/GetObjectScale`
    * Object segmentation: `simSet/GetSegmentationObjectID`
    * List all the objects in the scene: `simListSceneObjects`.
    * Domain Randomization#: `simSetTexture`
- Race API:
  - Starting and resetting of a race: `simStartRace, simResetRace`.
  - Tracking of race progress: `simGetLastGatePassed, simIsRacerDisqualified`
  - Get collision information: `simGetCollisionInfo`
  - Disabling the generation of race log files: `simDisableRaceLog`.
- Utilities:
  - Environment Geometry API#:

189

* Voxel grids and Signed Distance Field construction: `buildVoxelGrid/buildSDF`
* Occupancy and signed distance for points: `isOccupied, getSDF`
* Signed distance field gradients: `getSDFGradient`
* Project to nearest free space: `getNearestFreePt`
  - Camera Model API[#]:
    * Change camera field of view: `camSetFoV`
    * Change camera distortion parameters: `camSetDistortionParams`
    * Enable rolling shutter effect: `camEnableRollingShutter`
  - Event camera API[#]:
    * Generate event data: `generateEvents`
    * Set event camera parameters (contrast threshold, noise etc.): `camSetEventParams`
* Visualization API[#]:
  - Suite of functions aimed at plotting for debug purposes: `simPlotPoints, simPlotMarkers, simPlotString, simPlotTrajectory, simPlotTransform`
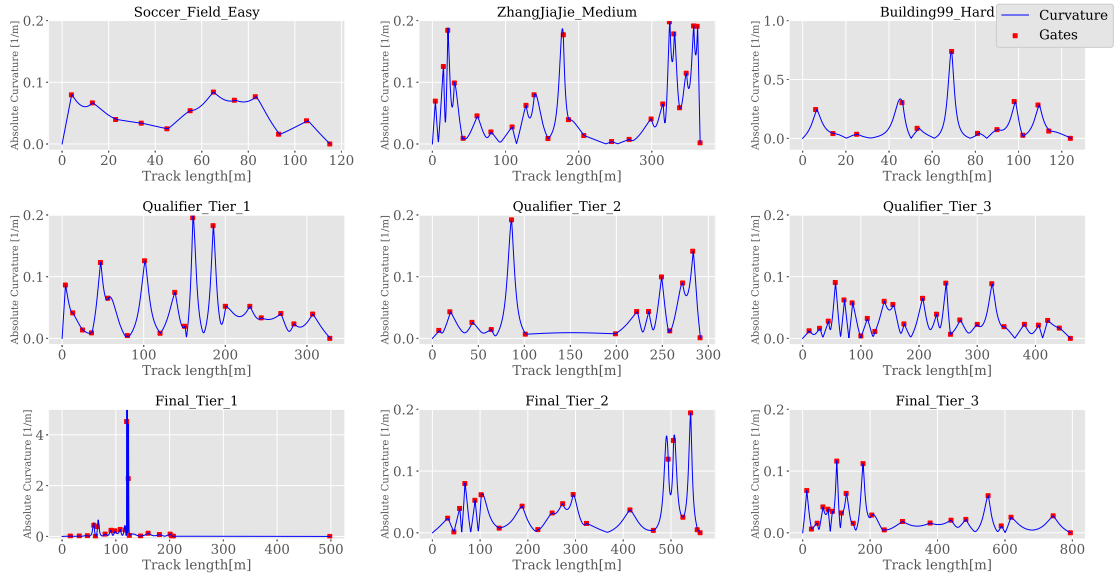
Appendix B. Race track complexity

The race track complexity metrics for all tracks are visualized in Fig. 6. Fig. 6a visualizes how the curvature changes along each race track, and Fig. 6b visualizes how the gate visibility (number of image pixels occupied by the gate divided by total number of pixels) varies along each track.
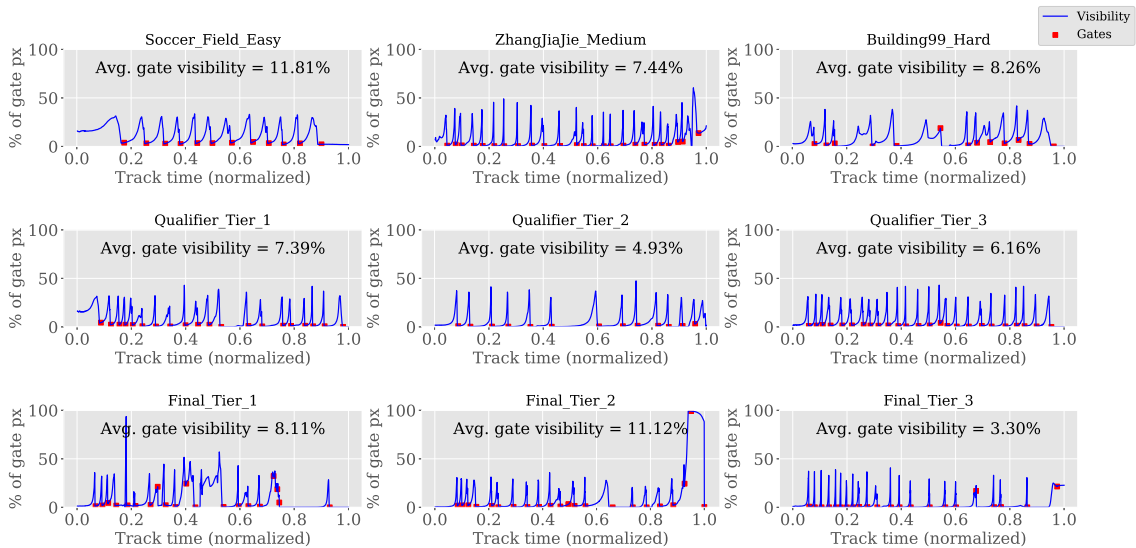
Appendix C. Events and optical flow generation

For simulating events from the RGB image stream of a camera, we loosely follow the framework presented in Rebecq et al. (2018). As an improvement to the above approach, we utilize Unreal Engine's graphics capabilities directly for improved performance. Pixel intensity differences are first computed through image subtraction, which is written as a shader that runs directly on the GPU (thus natively parallelized). This allows for calculation and rendering of pixel differences at near-realtime frame rates. This representation is then converted into simulated asynchronous event data (which is in process), again by utilizing compute shader nodes written in High Level Shading Language (HLSL) to ensure high performance and tight coupling between the renderer and the event generation.

We also provide access to ground truth optical flow images, which can be used for learning motion models and pose predictions (Fischer et al. (2015), Maurer and Bruhn (2018)). In the simulator, we obtain optical flow information by accessing Unreal Engine's rendering pass, which also encodes screen space velocities for all pixels; information that is subsequently used also to compute motion blur.

(*a*) Track complexity for planning and control tasks, measured in terms of instantaneous curvature. Note the different scales on X and Y axis.

(*b*) Track complexity for perception tasks, measured in terms of next gate visibility in the image. Note that each track is of different length, and the X-axis in is normalized.

Figure 6: Track Complexity