Distribution Statement A: Approved for public release. Distribution is unlimited

Unreal Engine-Based Photorealistic Aerial Data Generation and Unit Testing of Artificial Intelligence Algorithms

Andrew Buck[a], Raub Camaioni[b], Brendan Alvey[a], Derek T. Anderson[a], James M. Keller[a], Robert H. Luke III[b], and Grant Scott[a]

[a]Department of Electrical Engineering and Computer Science, University of Missouri, Columbia MO, USA
[b]U.S. Army DEVCOM C5ISR Center, Fort Belvoir, VA, USA

ABSTRACT

A number of real-time object detection, tracking, and autonomy artificial intelligence (AI) and machine learning (ML) algorithms are being proposed for unmanned aerial vehicles (UAVs). A big challenge is can we stress test these algorithms, identify their strengths and weaknesses, and assess if the UAV is safe and trustworthy? The process of collecting real-world UAV data is costly, time consuming, and riddled by lack of quality geospatial ground truth and metadata. Herein, we outline a fully automated framework and workflow to address the above challenges using free or low-cost assets, the photorealistic Unreal Engine (UE), and AirSim aerial platform simulator. Specifically, we discuss the rapid prototyping of an outdoor environment combined with the robotic operating system (ROS) for abstracting UAV data collection, control, and processing. Real and accurate ground truth is collected and metrics are presented for individual frame and entire flight collection evaluation. Metrics recorded and analyzed include percentage of scene mapped, 3D mapping accuracy, time to complete task, object detection and tracking statistics, battery usage, altitude (from ground), collisions, and other statistics. These metrics are computed in general and with respect to context, e.g., clutter, view angle, etc. Overall, the proposed work is an automated way to explore UAV operation before real-world testing or deployment. Promising preliminary results are discussed for an outdoor environment with vegetation, short and long range objects, buildings, people, vehicles, and other features for a UAV performing loitering and interrogation.

Keywords: artificial intelligence, machine learning, detection, tracking, autonomy, simulation, Unreal Engine, ROS, AirSim

1. INTRODUCTION

Unmanned aerial vehicles (UAVs) have become widely used as research platforms for developing artificial intelligence (AI) algorithms. UAVs can move freely in space and utilize a variety of sensing modalities, making them well-suited for studying autonomous behaviors. Furthermore, there are many application domains that would benefit from imbuing UAVs with some degree of autonomy. These include tasks such as 3D scene reconstruction, surveillance and reconnaissance missions, explosive hazard detection, search and rescue, and many others. Even providing a UAV with some limited amount of automation can benefit human operators, freeing them to focus on other tasks.

Developing AI systems for UAVs can be a challenging endeavour. Collecting data in real-world settings can be costly and time consuming. The lack of well-annotated ground truth creates an additional burden of data cleaning and validation. In many cases, such as 3D reconstruction, there is no available ground truth that can be used to verify the accuracy of algorithms in development. For applications like object detection, a human needs to manually label (or use automated tools to assist with labeling) each frame of imagery. Autonomous behaviors

Send correspondence to Andrew Buck
Andrew Buck: E-mail: buckar@missouri.edu

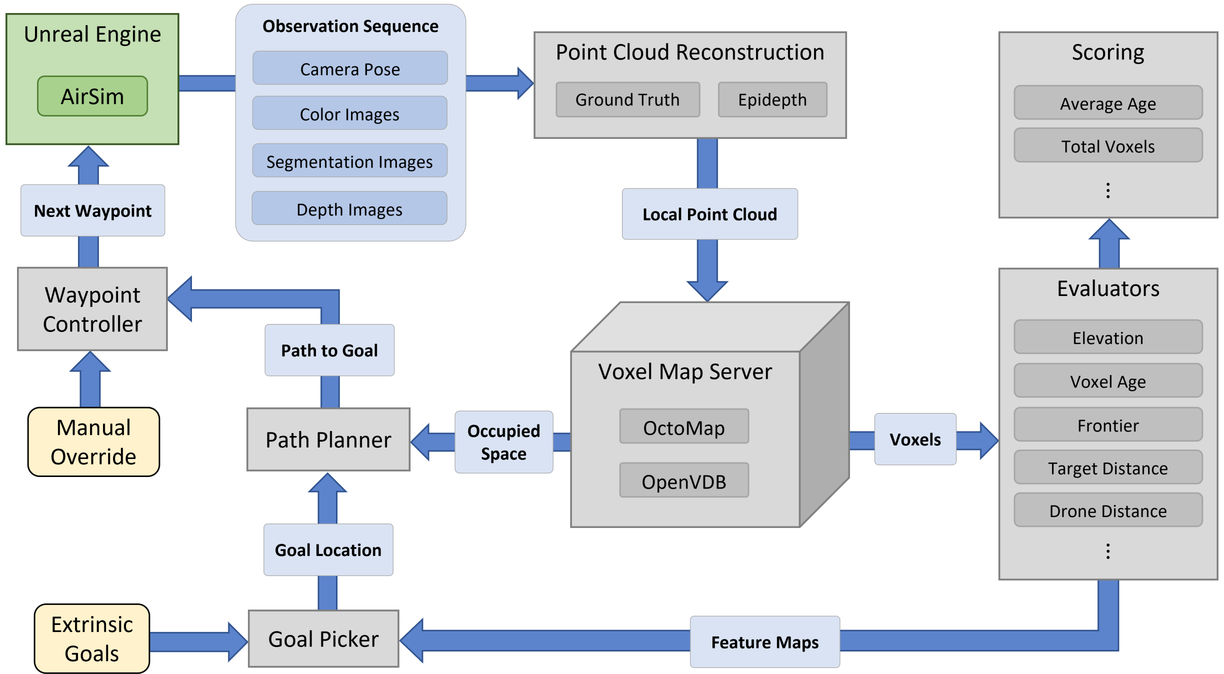

Figure 1: Overall simulation architecture.

tested on real UAVs can fail in unexpected ways, causing physical crashes and potentially dangerous outcomes. Obviously real-world testing must occur at some point before deployment, but a large part of development can benefit from the use of sophisticated simulation tools.

In recent years, several tools have matured to the point where it has become feasible to perform many of the tasks that go into developing AI systems for UAVs using relatively low-cost hardware and software. Unreal Engine[1] (UE) provides a photorealistic simulation environment that runs on a typical high-end desktop computer and has a large marketplace of maps and assets that can be used to create customized virtual worlds. The AirSim[2] extension to UE is an open-source library that gives developers a simple and straightforward API to control a drone platform and to capture telemetry and image data. One way to connect to AirSim is with the robotic operating system (ROS),[3] which provides many tools and packages for working with mobile robots. ROS is a distributed system that provides a message passing framework between computation nodes, allowing individual algorithm components to be abstracted and isolated from the rest of the system. This approach has several benefits, including the ability to "swap out" components to evaluate different ideas or algorithms. For example, a depth estimation algorithm could be replaced with ground truth depth from the simulator to focus only on the autonomous behavior of the system. This "unit testing" capability gives a way for AI practitioners to rapidly prototype new ideas without needing to rebuild the entire system.

Working in simulation has several additional benefits over performing experimentation in real-world settings. Ground truth is readily available, including camera depth images, object segmentation, and drone pose. This can be used to verify the accuracy of algorithms such as depth estimation or object detection. Simulated data can also be used to augment real data sets, often resulting in increased performance.[4,5] We have been able to show how working with simulated data can improve workflows for many UAV applications[6] and be deployed on real hardware.[7]

In this article, we present a simulation framework to develop and test AI algorithms for UAVs using UE, AirSim, and ROS. We utilize a spatial knowledge map, stored in the form of a voxel grid, which can be sampled

Figure 2: Mountain Village simulation environment rendered in Unreal Engine.

and evaluated to determine both where the UAV should go and overall run performance. Our method is modular and adaptable to many different application scenarios, of which we demonstrate both “loiter” and “search and rescue” examples. The runs are recorded and shown with several visualizations, and some initial scoring metrics are given. We conclude with a discussion of the current state of the framework and plans for future work in this area.

2. SIMULATION ARCHITECTURE

In this section, we describe the overall simulation architecture used to evaluate the autonomous operation of UAVs with Unreal Engine and the ROS framework. An overall system diagram is shown in Figure 1. The flow of data and control operations is discussed below, starting with how a sequence of observations is captured from a simulated drone in Unreal Engine using AirSim and published to the ROS network. These observations are collected and assembled into a model of the environment represented by the map server. Various features and scoring metrics can be computed on this model, providing attributes to a module that picks the next goal location for the drone and plans a path to reach it. This plan is then implemented by the control module, which sends commands back into Unreal Engine via AirSim and instructs the drone where to go next.

2.1 ROS Framework

The backbone of our simulation framework is the ROS[3] network, which is used to pass messages between independent control modules. Each module runs as a node on the network with its own thread of execution, resulting in a highly parallel and possibly distributed architecture. This is an important aspect to model, since real-world operations may take place on a variety of network-connected hardware of varying computational capability. It also allows for individual modules to be replaced with with different algorithms, providing a way to isolate and test various parts of the overall system.

The ROS ecosystem consists of nodes that act as message publishers and subscribers. Each node can subscribe and publish to individual message topics. A topic is declared to be of a specific message type, which can include pose information, camera parameters, image data, point clouds, command strings, and a multitude of other data types. All of the message types used in this work are standard types included in the ROS system libraries. Additional custom data types can be defined, however we have found the basic types to be sufficient for developing a rapid system prototype. The following sections describe in detail the various computational nodes that connect to ROS and the message types that are passed between them.

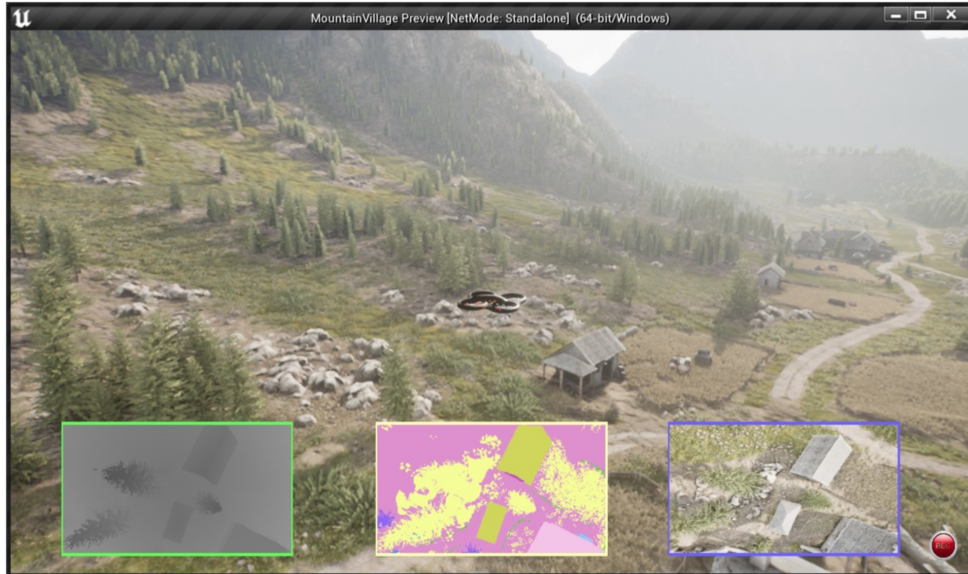

Figure 3: AirSim running in Unreal Engine, simulating a drone in flight. The subwindows show nadir camera views: (from left to right) depth, segmentation, and color images.

2.2 Unreal Engine

We use Unreal Engine[1] as the simulation environment for our work. We chose UE over other alternatives because of its state-of-the-art photorealistic rendering capabilities and the vast community of creators and developers that contribute to the UE ecosystem. This makes it exceptionally easy to acquire pre-built 3D assets such as landscapes, buildings, vehicles, people, and many other types of content. Many publishers make available completely modeled environments, requiring little end-user effort to stand up a simulation in a new scene. Alternatively, UE provides the flexibility to create custom environments with a combination of assets from different creators, providing generalization to a wide variety of applications.

Figure 2 shows an example scene that we have used to develop and test our simulation framework. It consists of approximately 1 sq. km. of a mountain valley environment and includes forests, roads, and village buildings.[8] This environment is well-suited for testing our UAV autonomy algorithms, as it provides ample space to explore while including both natural and man-made objects. The scene also has several disjoint building clusters connected by an irregular road network, which gives variation and also several points of reference to use for describing mission scenarios.

2.3 AirSim

AirSim[2] is a plugin for Unreal Engine that enables the simulation of a physical drone using the same control logic that would be applied on a real-world UAV. It provides a Python scripting interface, as well as direct ROS integration. The simulated drone can be configured with multiple virtual cameras, each with independent intrinsic and extrinsic parameters and capable of collecting color, segmentation, and depth images. The drone responds to control commands that direct it to move to a specific location and orientation, relative to the world origin. Other capabilities, such as direct velocity control and weather effects, are available but not used in our framework. Figure 3 shows the AirSim main window with nadir camera views of depth, segmentation, and color while simulating a drone in flight.

One major advantage of using a simulated environment to develop algorithms and test capabilities is the ability to record ground truth information that is not available in real world settings. For instance, depth information may not be available with the accuracy provided by the simulator, or in some cases it may not be available at all. We can use the known ground truth depth to evaluate the accuracy of a depth estimation algorithm or to

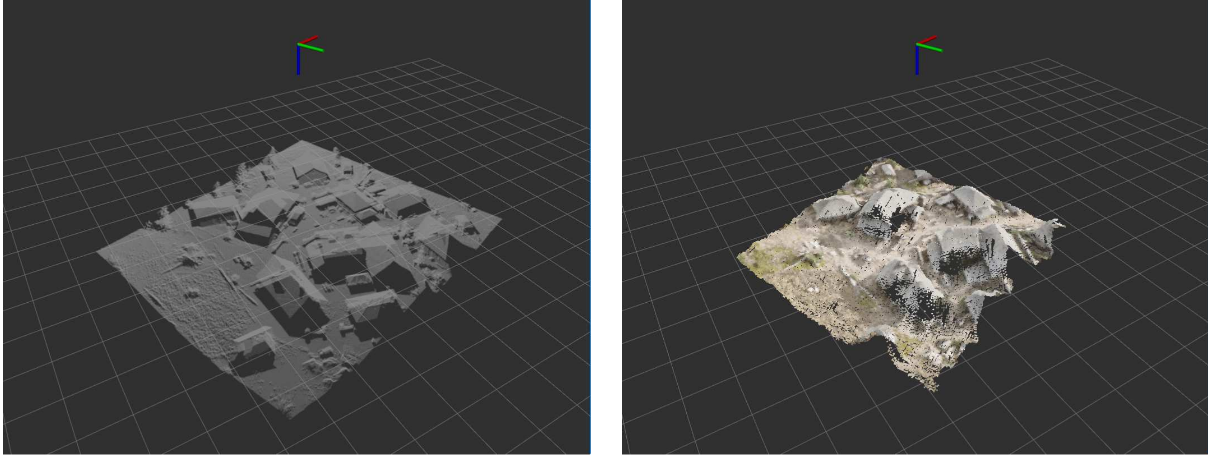

Figure 4: A local point cloud reconstruction from the nadir camera. The left image shows the ground truth constructed using the depth map from AirSim and the right image shows the output of the Epidepth algorithm, which uses a pair of color images and their relative poses to reconstruct the 3D points.

allow the environment to be sensed with perfect accuracy so that other aspects of the autonomy stack can be assessed. Ground truth position information provided by AirSim can also be used to aid in reconstruction or to evaluate overall performance. It should be noted that although perfect sensor information can be provided by the simulator, real-world performance may depend on the ability to handle imperfect information. AirSim and our simulation framework can be configured to provide degraded sensor information to study the impact of sensor noise in these situations.

2.4 Point Cloud Reconstruction

A fundamental aspect of our simulation framework is the ability to construct and utilize a 3D map of the environment. One common representation for such a map is an occupancy grid, where each grid cell indicates if a region of space is free or occupied. This will be discussed in detail in the following section, but this type of map can be constructed by aggregating individual point cloud observations, where each local point cloud contains the 3D projected image from the current camera view. Although any camera view can contribute to the overall map, our work focuses on using the nadir camera for point cloud reconstruction and map building.

The local point cloud can be obtained either as ground truth using the depth map from AirSim, or using a depth estimation algorithm with the sequence of color images and the drone poses. Figure 4 shows two local point clouds constructed using both of these methods. In both cases, the point cloud is built from the nadir camera looking downward and uses a local reference frame with real-world units. The point cloud is localized to the world coordinate system using the known camera pose, provided by AirSim. Note that we are not considering the localization problem of a full SLAM system, instead assuming that the world pose of the camera is known (to some degree of accuracy).

In the first approach (Figure 4 left), the point cloud is built using the synchronized ROS messages indicating the nadir camera pose and the depth image. From this information, each image pixel can be projected out from the camera origin by the distance indicated in the depth image, resulting in a dense local point cloud. Using this ground truth point cloud, we can construct very accurate maps and directly evaluate the accuracy of a reconstruction algorithm.

The second approach (right) uses the Epidepth algorithm[9] to estimate the image depth from only a pair of color images and their poses. Using stereo reconstruction techniques, the two images can be aligned such that depth can be computed by determining the disparity between two corresponding points in both images. This results in a dense point cloud of estimated depths from one of the camera poses. Most pixels that can be matched between images are projected as 3D points, however some gaps can occur due to occlusion or noise. To

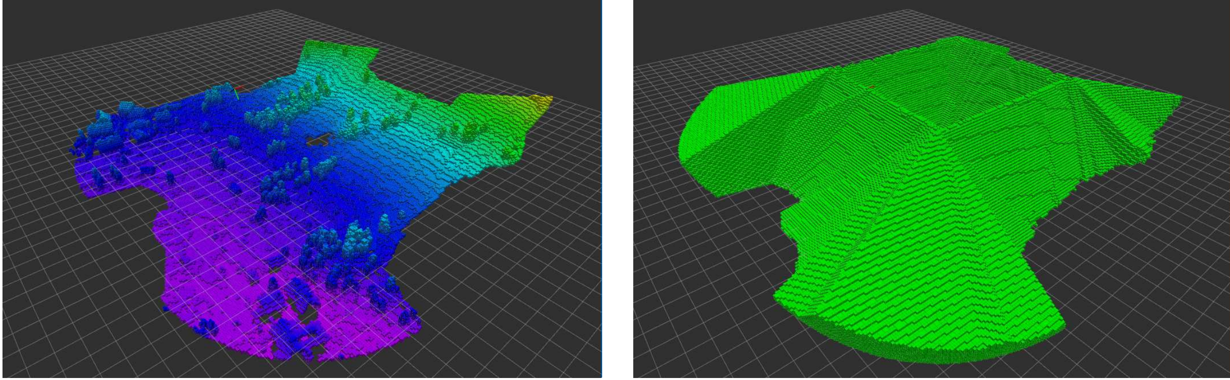

Figure 5: An environment map built using OctoMap from a drone flying in a square pattern. The left image shows the set of occupied voxels, colored by elevation and the right image shows the voxels representing observed free space. Grid cells that are not part of either set are unknown.

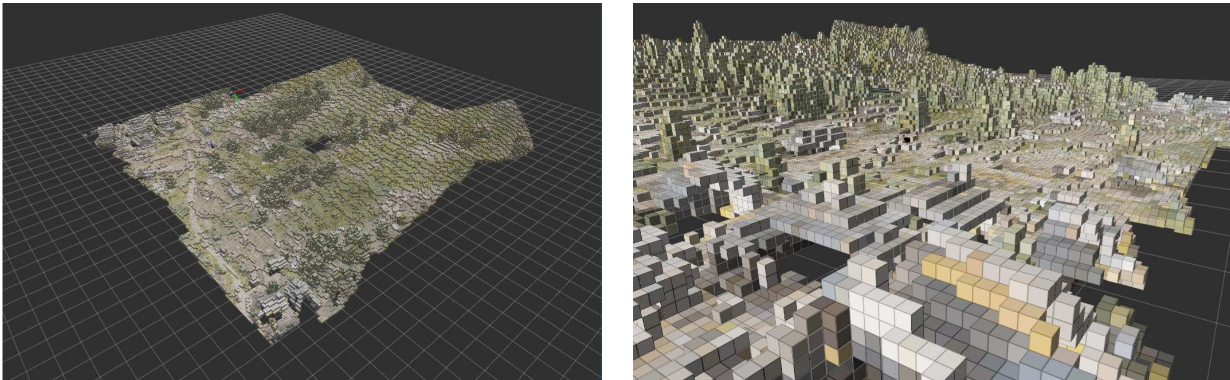

Figure 6: An environment map built using OpenVDB from a drone flying in a square pattern. The left image shows an overview and the right image shows a closeup. Each voxel here stores a color attribute.

ensure a high degree of accuracy, the algorithm can choose to only accept image pairs where the pose difference is within a certain tolerance threshold. Because of this, the rate at which estimated point clouds are computed may be less frequent than the ground truth point clouds. We therefore often choose to use the ground truth point clouds when evaluating certain aspects of our overall autonomy stack, although we maintain compatibility with the Epidepth approach and use it when appropriate.

2.5 Map Server

Individual point cloud observations are generated from the current drone position, but after the drone leaves an area, the knowledge that it has gathered needs to be saved in a persistent way. We use an occupancy grid approach that stores information about each grid cell in a map that is updated and queried as the simulation progresses. For 3D environments, we define a voxel size to be used for the map and establish a world coordinate system over the entire simulation space. As new point clouds are observed, they are integrated into the existing map using the coordinates provided by AirSim. The map server then provides access to the map data for other modules, such as feature computation and path planning. We have explored two approaches for representing the world map: OctoMap and OpenVDB.

OctoMap[10] is a hierarchical and probabilistic 3D voxel mapping solution based on octrees. It is commonly used with ROS and has pre-built packages that make it easy to set up. The launch configuration specifies the voxel size and the topic name of the incoming local point clouds. When a new point cloud is received by

OctoMap, it casts a ray from the point cloud origin (set as the camera focal point) to each point and determines all intersecting voxels. These are then marked as free space in a probabilistic way, and the voxel containing the point is marked as occupied. By updating the probability that a grid cell is occupied rather than using a binary value, OctoMap is less sensitive to errors in the point cloud reconstruction and can also respond to dynamic scenes. Figure 5 shows a voxel map built using OctoMap, where a threshold has been applied to split the set of observed grid cells into occupied and unoccupied sets.

An alternative voxel map representation can be built using OpenVDB.[11] OpenVDB is a software library for storing and manipulating sparse 3D data structures. It efficiently handles large volumes of data and has been used for simulation and effects in the feature film industry. We use OpenVDB as a voxel map server that aggregates local point cloud observations into a combined world map. Unlike OctoMap, there is no built-in handling of probabilistic occupancy, however certain operations that require iterating over the entire set can have better performance due to the underlying storage method. Figure 6 shows a voxel map using OpenVDB, where each voxel has a color attribute obtained from the point cloud observations.

2.6 Evaluators

The map that is constructed by the map server represents the agent's current knowledge of the environment. There are many ways the agent can utilize this knowledge to choose future actions. We take the approach here of defining 2D feature maps that provide the agent with processed information pertaining to the goals of navigation and exploration. In this way, we assume that the environment is more interesting horizontally than vertically. We also scale each feature map into the unit interval, with values closer to 1 generally representing locations that are more desirable to explore. Figure 7 shows several of the feature maps that we use in our simulation framework. This set of features is not exhaustive and many others could be computed. Using the voxel data from the map server, we define the following 2D feature maps that can be used by other modules.

- **Elevation:** The elevation map is a top-down 2D projection of the observed voxels where the value of each pixel coordinate corresponds to the maximum occupied voxel index in that vertical column. This essentially creates a digital elevation model (DEM) of the environment that is dynamically updated.
- **Age:** Each time a voxel in the map is updated, we save the current timestamp as an attribute of that voxel. This can then be used to compute the voxel age. Voxels that have not been observed in a long time will have a greater age than those that have been observed more recently. We may apply an exponential decay function to map the values into the unit interval, such that voxels with value 1 have just been observed and those with lower values are older. We create a 2D feature map by taking the maximum age score along each vertical column, which gets the age values of the most recently observed voxels at each location.
- **Frontier:** When exploring an unknown area, an autonomous agent might seek to expand the frontier of the observed map. We can compute this from the map server by identifying all of the boundary pixels on a 2D projected map of the occupied voxels. Any pixel that has an unobserved neighbor is set to 1 and all others are set to zero. We then compute the Euclidean distance transform using this binary mask, measuring the distance to the nearest unobserved pixel from each pixel in the image. This distance is inverted and scaled to the unit interval so that locations with value 1 are on the edge of the explored region and interior locations have smaller values.
- **Target Distance:** When running an autonomous scenario, there is often some location that is designated as the target location. This may be the origin point if the objective is to map out the local region around the takeoff point, or a remote location if the objective is to perform reconnaissance on some other area of the map. The target distance feature map is computed by measuring the distance to the defined target location and then inverting and scaling the output into the unit interval such that value 1 is directly at the target location with smaller values farther away. For some applications, it may be appropriate for the target location to be a region rather than a single point. In this case, the target distance feature is the distance to the nearest point within the target region.

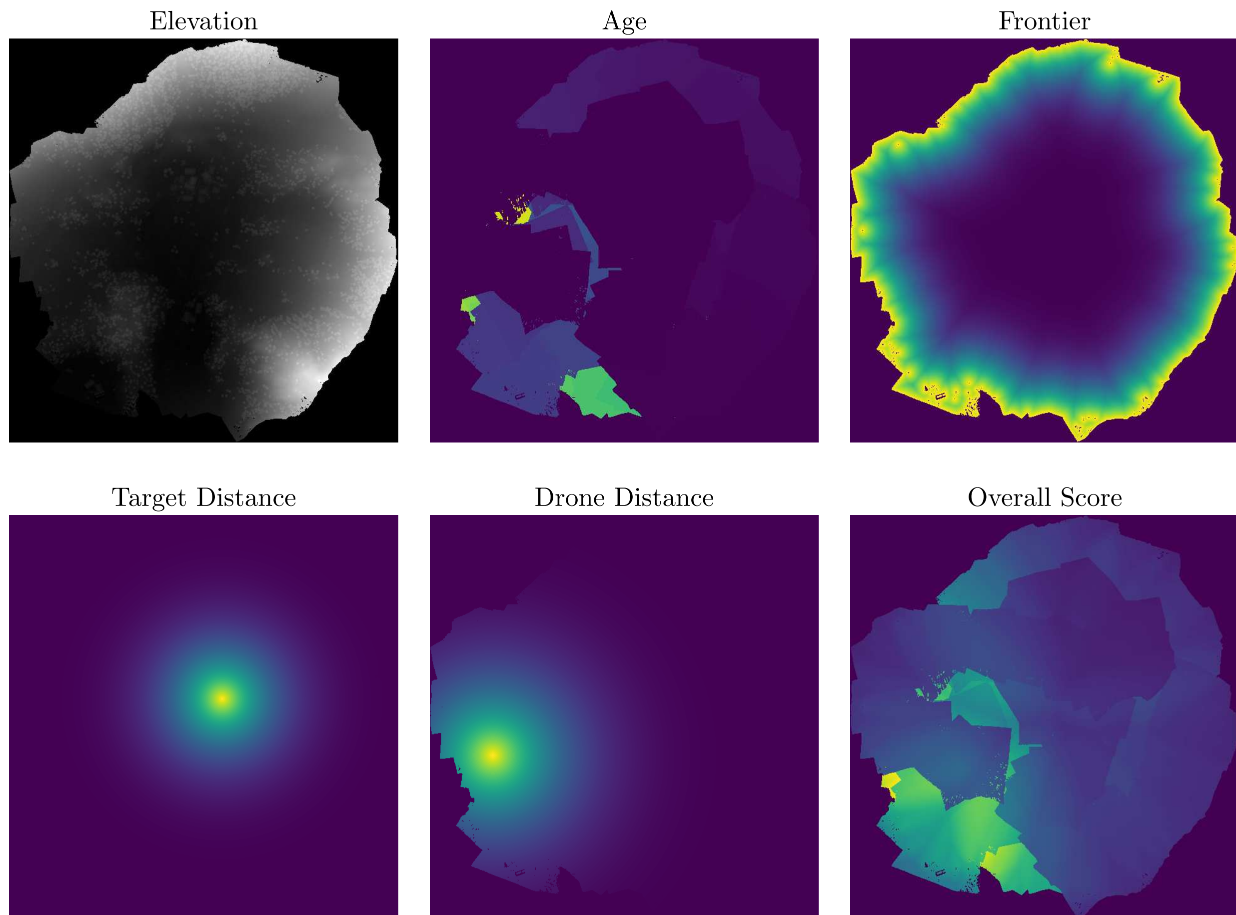

Figure 7: 2D feature maps computed from the voxel map provided by the map server module. The elevation feature can be used for navigation, whereas the others are scaled to the unit interval and are used in choosing where to explore.

- **Drone Distance:** We often need to evaluate locations based on how far away they are from the current drone location. The drone distance feature map is computed in the same way as the target distance, using the drone location as the target. A value of 1 represents the current drone location in the map with values decreasing farther from the drone.

2.7 Goals

An agent can act autonomously within our simulation framework by selecting its own actions, independent of direct user input. In the context of a UAV exploring an environment, actions are typically expressed as moving the drone to a new pose. At a high level, we can define a goal location for the drone to move toward and use lower level modules to actually plan a path and control the motors. We utilize a multi-criteria decision-making (MCDM) approach using the feature maps defined above to evaluate the known environment and select a goal location.

For each feature map $k \in [1, ..., N]$ containing pixels $p_{ijk} \in [0, 1]$, we define a weight $w_k \in [0, 1]$ indicating that feature's relative importance. A weighted sum of the feature maps then gives the overall score image, where the score of each pixel location is computed as

$$S(p_{ij}) = \sum_{k=1}^{N} w_k \cdot p_{ijk}. \tag{1}$$

An example is shown in Figure 7, which weights the age, frontier, target distance, and drone distance features equally. Within this image, the pixel with the greatest value is chosen as the next goal location. This location can change with each new observation as the features are recomputed.

Each of the above features contributes to the selection of the goal location. The age feature guides the drone toward regions of the map that have not been observed recently, ensuring that the map stays up to date. The frontier feature pulls the drone toward the edge of the explored space. The target distance feature ensures that the drone does not move too far away from the specified target location, and the drone distance feature helps select goals that are nearby, rather than on the opposite side of the map. Together, and with user-defined weights, they provide a way to establish an autonomous behavior that continuously operates in an environment.

2.8 Path Planner

Given a goal location, the path planner module is responsible for planning a route to the goal from the current drone position. A simple version of this module uses the elevation map to determine the elevation of the goal and the minimum required elevation to travel there in a straight line. An additional height buffer can be added to ensure that collisions are avoided.

A more advanced path planning module could utilize the OctoMap map server to find a route that stays within known free space. If navigating to the goal requires moving into unexplored regions of the map, the path planner could determine an appropriate route and elevation profile, and make any necessary adjustments as the plan is carried out and more of the map is observed. An even more sophisticated path planning module could utilize additional feature maps or user preferences to avoid certain areas or to stay within a given boundary.

2.9 Waypoint Controller

The waypoint controller module performs the communication with AirSim to actually fly the drone. We use the high-level AirSim API to specify a target position as a waypoint and let the built-in flight controller handle the low-level control. In this way, we only need to specify the current goal position computed using the MCDM method and the path planner. The low-level controller can optimize a trajectory along a sequence of waypoints, so the path planner module can optionally provide additional future waypoints if so desired.

The control module also acts as a state machine to provide different modes of operation. A manual control mode can be initiated to immediately take over the drone control in case of emergency. Other behavior patterns, such as pre-scripted flight sequences, can also be activated through this module. Usually, the drone operates in the autonomous mode using the method described above, although some maneuvers like takeoff, return to home, and landing may override this behavior.

2.10 Scoring Metrics

As the simulation progresses, we keep track of several different scoring metrics that can be used to evaluate the performance of the agent and compare various behaviors. These provide a way to quantify the performance of any given run of an algorithm within the simulation framework. In general, we seek to capture aspects of the system that are deemed relevant to human operators and can provide insight into how the useful the system is for a particular task. Most scoring metrics are computed from the evaluated features, but some may use information directly from AirSim to gain access to the underlying simulation statistics and ground truth.

Evaluating the quality of a 3D map is essential for reconstruction methods and may be the primary scoring metric for some applications. Using a purely simulated environment gives access to the underlying ground truth so that these methods can be evaluated directly. We can assess the accuracy of individual point cloud reconstructions estimated by algorithms such as Epidepth by comparing against the the depth image provided by AirSim. The average number of pixels that are estimated to have a depth within a given tolerance of the known ground truth is one such accuracy metric that can be computed. On a broader scale, the voxel map accuracy

can be compared to known ground truth by computing the set intersection and union of the ground truth and estimated voxel sets. We note that the voxel map computed from ground truth point cloud observations will likely be a closer ideal to what can be achieved by a depth estimation algorithm as compared to a voxel map exported from the simulation environment directly (such as by using mesh geometry) since this can include occupied voxels in unobservable regions (such as below the terrain).

Other statistics can be computed from the voxel map provided by the map server to gauge how effectively a given UAV behavior achieves its goals. These may include the number of voxels observed (or a percentage of the known environment area), the average age of the observed voxels (time since last observation), time to complete a given task (find a specific target or reach a given location), battery usage, elevation profile, and if there were any collisions. The overall performance of an autonomy algorithm is largely dependent on the given task, and each application may have a unique set of ways to evaluate a run within the simulation.

3. EXAMPLE SCENARIOS

In this section, we present two example scenarios that demonstrate the capabilities of our simulation framework. The first is a loiter scenario designed to map and observe the local region around the takeoff location. The second is a search and rescue mission designed to show how the framework can be used to accomplish remote tasks. They both utilize the architecture described above with different feature weights and goal specifications. While these two examples are not the only ways this framework can be utilized, they demonstrate general mission scenarios that can be adapted and customized to work for many real world applications.

3.1 Loiter

In this first scenario, the objective is to build a map of the local region and keep it up to date. This can be used to do initial 3D reconstruction or to maintain automated overhead surveillance of an area. The drone is configured with a downward-looking (nadir) camera in AirSim that provides both color and depth imagery. Map building can be done using either the Epidepth estimation algorithm or known ground truth. Using a depth estimation algorithm allows the experimenter to focus on 3D reconstruction accuracy, while using the simulator to provide depth allows for focus on exploration and autonomous behaviors. We use the feature maps described above to define how the drone explores the environment. The flight behavior is recorded and we can qualitatively and quantitatively examine the performance of any given run.

A snapshot of the simulation in progress for the loiter scenario is shown in Figure 8. The simulation takes place across the ROS network, and can be visualized and analyzed in a variety of ways. Here, we show several windows that provide insight into how the system is operating. The top-left window is the AirSim simulation running in Unreal Engine, which provides the simulation environment and ground truth. The subwindows show nadir camera views for depth, segmentation, and color images. The top-right window shows the point cloud reconstruction occurring on the current frame using both the known ground truth (gray) and the Epidepth estimation (color). This is used to construct the voxel map shown in the bottom-right window, which represents all of the information that the drone has gathered so far. We use OpenVDB with the ground truth to build the map in this example, but Octomap and Epidepth (or other methods) could be used as well. The voxel map is projected into 2D feature maps, including the elevation map (top-center) and the voxel age, frontier, origin distance, and drone distance features. These four features are weighted equally and combined to give the overall score map shown in the bottom-center window. At any point during the simulation run, the map location with the current highest score in this image is chosen as the next waypoint for the drone. The drone controller plans a direct flight path to that location, looking in the forward direction and selecting an elevation 50 meters above the known elevation at that point. As the simulation progresses, we measure the average age of the voxels in the map and the total voxel count and plot these in the bottom-left window. We see that after some initialization period, the age score settles into a relatively steady value, even as the total voxel count continues to increase.

To gain a better understanding of the how the simulation progresses, Figure 9 shows a sequence of the elevation map and combined feature maps for this loiter scenario. When the simulation first starts ($t = 15$), the map shows only the area directly around the takeoff location and the area with the greatest score is close to the center of the map. Soon, the frontier score and age score for areas around the edge of the map begin to outweigh the origin and drone distance features, which work to keep the drone near the center of the map. As

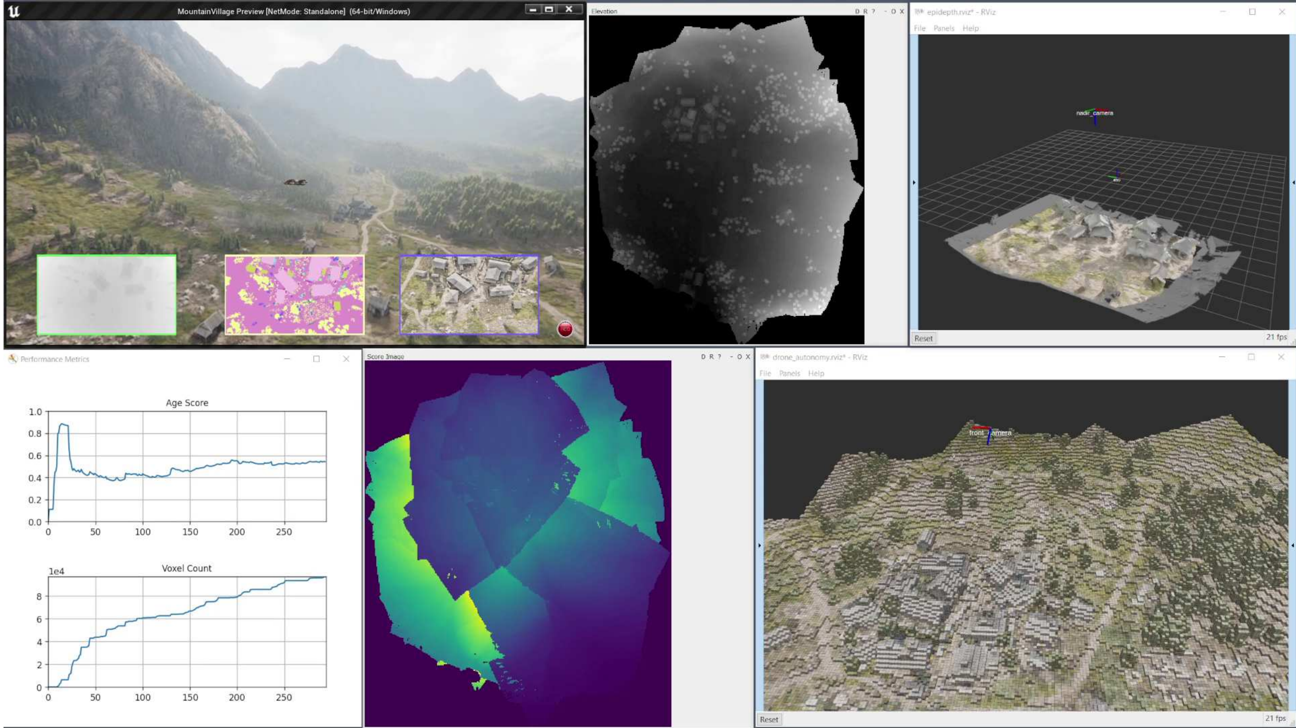

Figure 8: A collection of windows showing the progress of a drone running the "loiter" scenario. Clockwise from top-left: AirSim running in Unreal Engine (simulation ground truth) with nadir camera subwindows for depth, segmentation, and color images; Constructed elevation map; Real-time assessment of the Epidepth depth estimation algorithm, showing the predicted point cloud in color against the ground truth point cloud in gray; Voxel map server running in OpenVDB, built from ground truth depth observations; Overall score image used to decide the next waypoint; Scoring metrics showing voxel age and count over time.

the drone moves toward the map edge and begins to collect new observations, the map expands and the features are recomputed. After a few iterations ($t = 45$) a small region around the origin has been observed and the age feature (along with the others) has determined that the region near the top of the map is the best area to explore next. Some time later ($t = 125$), the region in the lower-right has gone out of date and needs to be re-observed. In the last timestep shown ($t = 230$), the central region once again has been evaluated to be the best place to observe next. This process continues indefinitely, until stopped by the user or some mission-specific criteria has been met (timeout, percent of scene mapped, etc.). The four features work together to keep the drone anchored near the origin, while also exploring new areas around the map edges and not allowing the map to become outdated. These weights can be adjusted to give rise to new behaviors as demonstrated in the next example.

3.2 Search and Rescue

This second scenario demonstrates a search and rescue mission, in which a person has gone missing in an unexplored part of the environment and needs to be located. In this example, we assume that the only starting knowledge is that the person had last been seen to the south of the takeoff location. Figure 10 shows the simulation shortly after starting. Based on the given search direction, the target location is set far to the south, resulting in the southern edge of the map having the greatest overall feature score (bottom-left window) and being consistently chosen as the next place to explore. For this scenario, we only use the target distance, frontier score, and drone distance features, with the target distance receiving twice the relative weight as the other two

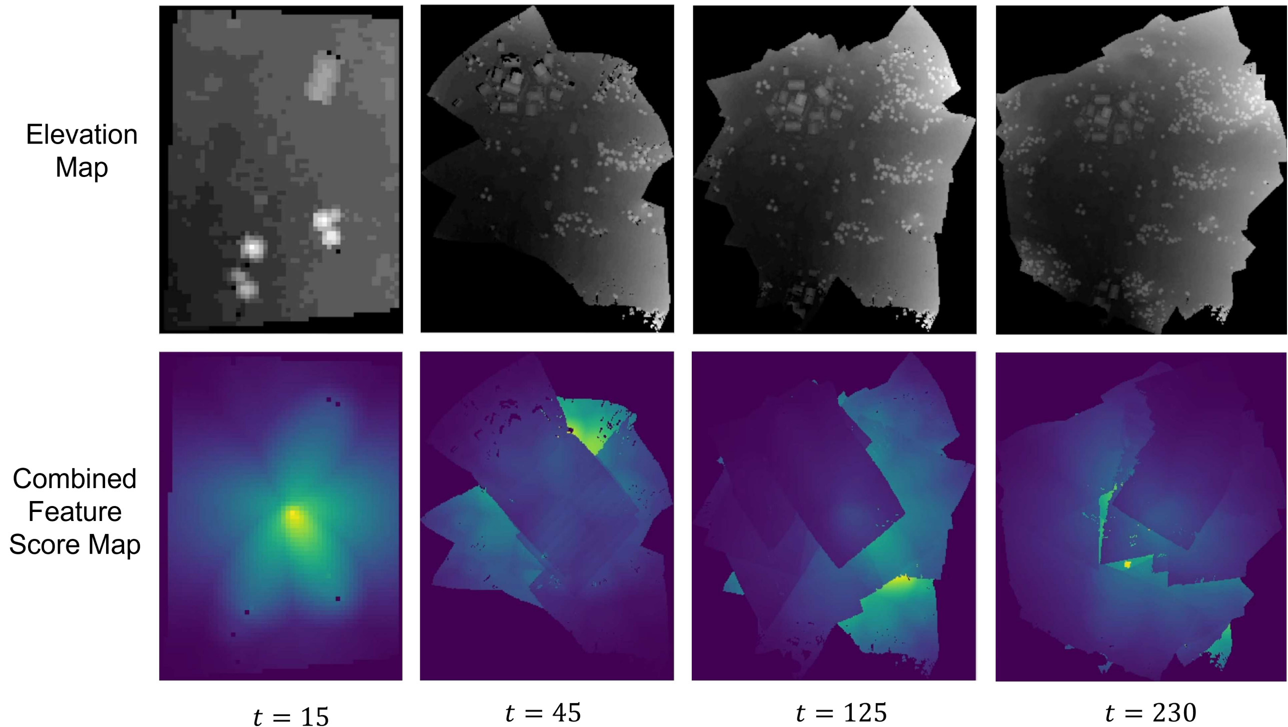

Figure 9: Selected timesteps from a sequence of elevation maps and combined feature score maps recorded during the "loiter" scenario.

features. (The voxel age and origin distance features would keep the drone near the takeoff location rather than exploring new areas of the environment and so are set to zero weight.) The bottom-right window shows the voxel map server using OpenVDB with full color attributes, and the bottom-center window shows the OctoMap implementation with color mapping to elevation. Both voxel maps are built using ground truth depth in this example. The white sphere in the map server windows shows the location of the current waypoint at the elevation of the observed map.

As the simulation progresses, the drone continues to fly to the south and construct a more complete map of the environment (Figure 11). Eventually, the drone reaches a ridge and identifies the person with the forward-facing camera. In our example here, the object detection is provided as ground truth from AirSim, registering a detection so long as the person is in frame and the drone is within 100 meters. As an additional level of realism, an actual object detection algorithm such as YOLO[12] could be applied to generate the detections. When a person is detected (and verified to be the person of interest) the target location feature changes to the person's location and the drone switches to a stop and stare behavior. This is to simulate the completion of the mission, at which point a human operator would take control.

This example shows how the simulation framework can be applied to a scenario where the target location may change over time. The goal selection module can utilize many sources of information, not necessarily only those present in the feature maps. For instance, when a target object is recognized or a certain condition is satisfied, the way that goals are chosen may change. Here, we adjusted the target location to match the detected person when found, but other behaviors could also be used to handle multiple targets or different mission constraints. Our use of ground truth data for this example allows us to focus on the goal selection and autonomous behavior aspect of the simulation without concern for reconstruction accuracy or detection performance. As these individual components mature, they can be integrated into the overall framework to evaluate their respective capabilities.

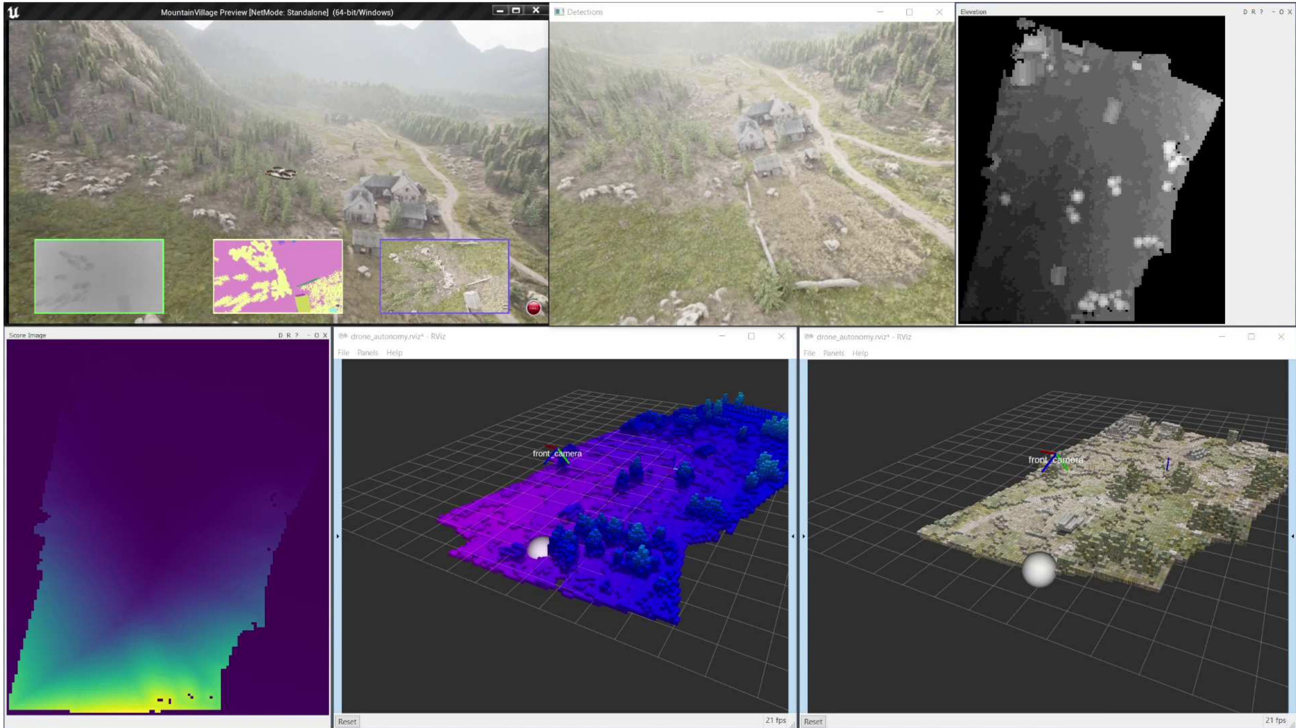

Figure 10: A collection of windows showing the "search and rescue" scenario shortly after takeoff. Clockwise from top-left: AirSim running in Unreal Engine (simulation ground truth) with nadir camera subwindows for depth, segmentation, and color images; Forward-looking camera running person detection; Constructed elevation map; Ground truth voxel map server running in OpenVDB showing color; Ground truth voxel map server running in OctoMap showing elevation; Overall score image used to decide the next waypoint. The white sphere in the map server windows shows the current waypoint location at ground elevation.

4. CONCLUSIONS AND FUTURE WORK

The simulation framework presented in this article demonstrates the potential of using simulated environments to develop and test AI algorithms for UAVs. The system can be extended with additional modules and evaluators that connect into the ROS network to achieve novel behaviors on new problem types not shown here. The general MCDM framework is flexible and can account for many different goals and objectives, balanced with user or system-defined weights that may adapt as conditions change.

More complex mission scenarios could work to incorporate a greater sense of human/robot teaming. This could involve recognizing a human agent within the scene and changing a target location based on the human's location. Multiple agents could share information with the map server, each utilizing a subset of the possible feature sets.

Future versions of the system will likely also take advantage of more sophisticated path planning algorithms. The basic single waypoint goal model implemented here could be extended to factor in known free space or other map features to plan a better route to a goal location. The simulation framework could also be used as the training environment for reinforcement learning applications.

In conclusion, there are many advantages to utilizing a simulation framework such as the one presented here for studying and prototyping systems that will eventually be deployed on real hardware. The simulation sandbox allows many ideas to be rapidly tested without risk of expensive failure. Many of the modules and ROS nodes can be transferred to using real sensors and flight data with little effort. The ability to "unit test" individual

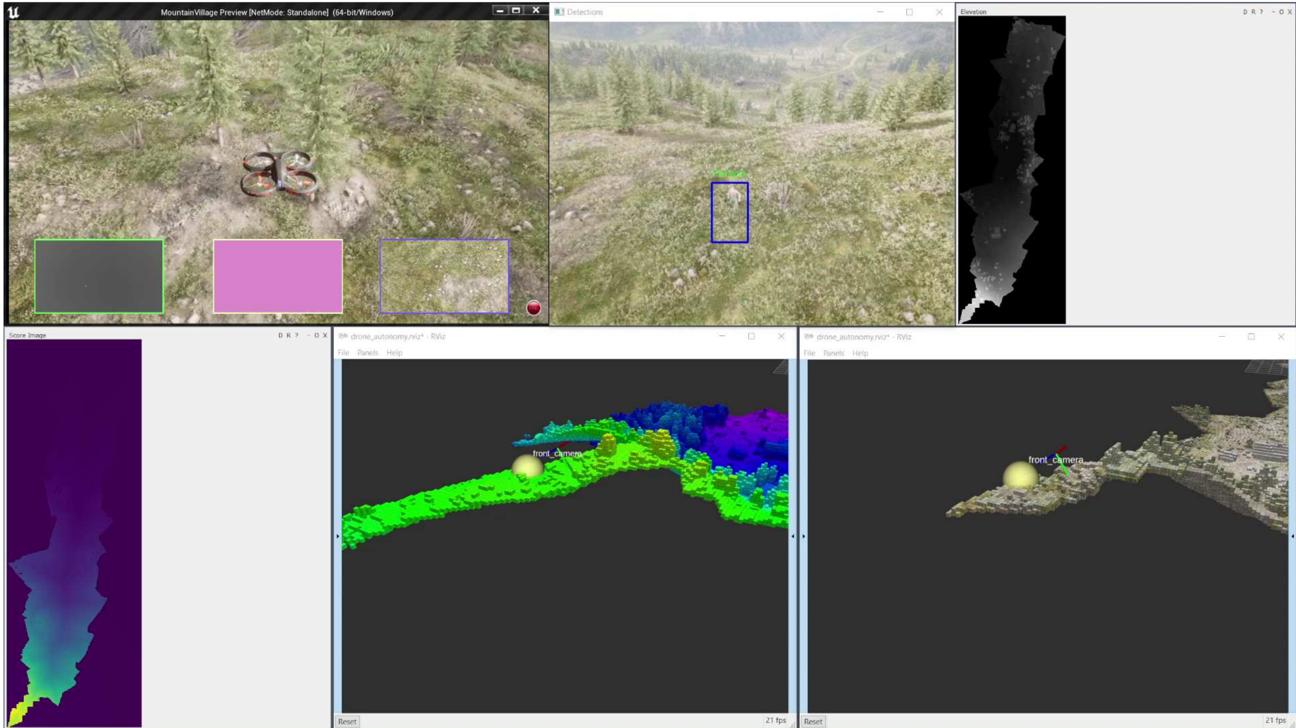

Figure 11: A collection of windows showing the "search and rescue" scenario after locating the missing person. The window descriptions are the same as Figure 10. Here, a person is identified in the forward-looking view (top-center) and the constructed maps show the southward flight of the drone.

system components allows algorithm designers to evaluate isolated parts of the complete autonomy stack with fine control over when to use ground truth and when to use real data. Overall, AI systems developed for for UAVs can benefit from the insights provided by simulation tools.

REFERENCES

[1] "Unreal Engine." https://www.unrealengine.com/. (Accessed: 8 March 2022).

[2] Shah, S., Dey, D., Lovett, C., and Kapoor, A., "Airsim: High-fidelity visual and physical simulation for autonomous vehicles," in [*Field and Service Robotics*], (2017).

[3] "Robot Operating System (ROS)." https://ros.org. (Accessed: 8 March 2022).

[4] Alvey, B. J., Anderson, D. T., Keller, J. M., Buck, A., Scott, G. J., Ho, D. K. C., Yang, C., and Libbey, B., "Improving explosive hazard detection with simulated and augmented data for an unmanned aerial system," in [*Detection and Sensing of Mines, Explosive Objects, and Obscured Targets XXVI*], Isaacs, J. C. and Bishop, S. S., eds., 10, SPIE, Online Only, United States (Apr. 2021).

[5] Alvey, B. J., Anderson, D. T., Yang, C., Buck, A., Keller, J. M., Yasuda, K. E., and Ryan, H. A., "Characterization of Deep Learning-Based Aerial Explosive Hazard Detection using Simulated Data," in [*2021 IEEE Symposium Series on Computational Intelligence (SSCI)*], 1–8 (Dec. 2021).

[6] Alvey, B., Anderson, D. T., Buck, A., Deardorff, M., Scott, G., and Keller, J. M., "Simulated Photorealistic Deep Learning Framework and Workflows to Accelerate Computer Vision and Unmanned Aerial Vehicle Research," in [*2021 IEEE/CVF International Conference on Computer Vision Workshops (ICCVW)*], 3882–3891 (Oct. 2021).

[7] Buck, A. R., Deardorff, M., Anderson, D. T., Wilkin, T., Keller, J. M., Scott, G. J., Luke, R. H., and Camaioni, R., "VADER: A hardware and simulation platform for visually aware drone autonomy research," in [*Unmanned Systems Technology XXIII*], Muench, P. L., Nguyen, H. G., and Skibba, B. K., eds., 15, SPIE, Online Only, United States (Apr. 2021).

[8] "Mountain Village Environment." https://www.unrealengine.com/marketplace/en-US/product/mountain-village-environment. (Accessed: 8 March 2022).

[9] Camaioni, R., Luke, R. H., Buck, A., and Anderson, D. T., "EpiDepth: a real-time monocular dense-depth estimation pipeline using generic image rectification," in [*SPIE*], (2022).

[10] Hornung, A., Wurm, K. M., Bennewitz, M., Stachniss, C., and Burgard, W., "OctoMap: An efficient probabilistic 3D mapping framework based on octrees," *Autonomous Robots* **34**, 189–206 (Apr. 2013).

[11] "OpenVDB." https://www.openvdb.org/. (Accessed: 8 March 2022).

[12] Redmon, J. and Farhadi, A., "Yolov3: An incremental improvement," *arXiv* (2018).