How Should Simulated Data Be Collected for AI/ML and Unmanned Aerial Vehicles?

Jeffrey Kerley, Derek T. Anderson, Brendan Alvey, and Andrew Buck

Electrical Engineering and Computer Science Department, University of Missouri, Columbia, MO 65211, USA

ABSTRACT

Large and diverse datasets can now be simulated with associated truth to train and evaluate AI/ML algorithms. This convergence of readily accessible simulation (SIM) tools, real-time high performance computing, and large repositories of high quality, free-to-inexpensive photorealistc scanned assets is a potential artificial intelligence (AI) and machine learning (ML) game changer. While this feat is now within our grasp, what SIM data should be generated, how should it be generated, and how can this be achieved in a controlled and scalable fashion? First, we discuss a formal procedural language for specifying scenes (LSCENE) and collecting sampled datasets (LCAP). Second, we discuss specifics regarding our production and storage of data, ground truth, and metadata. Last, two LSCENE/LCAP examples are discussed and three unmanned aerial vehicle (UAV) AI/ML use cases are provided to demonstrate the range and behavior of the proposed ideas. Overall, this article is a step towards closed-loop automated AI/ML design and evaluation.

Keywords: simulation, procedural, formal language, LSCENE, LCAP, AI, ML, artificial intelligence, machine learning, drone, unmanned aerial vehicle

1. INTRODUCTION

Current generation narrow versus general artificial intelligence (AI) is overly reliant on labeled data. But, where does the data and its truth come from? How large and diverse is the data and is it sufficient for achieving the task at hand? What is the overall cost for collecting and labeling the data? While older methodologies like unsupervised learning and newer strategies like self-supervised learning are being explored to address limitations with supervised learning, future AI will likely always be reliant on data with truth to some degree. In a similar vein to how data and high performance computing rejuvenated neural networks and help set deep learning (DL) in motion, simulation (SIM) is a powerful tool that can be used to help mitigate issues like the aforementioned. The current article is a step towards SIM for AI. Specifically, we explore the formal specification and procedural generation of a scene and collection.

Figure 1 illustrates our goal of closed-loop AI training and evaluation in SIM. The idea is an automated framework to learn a formal procedural language (P) to model a given task (T), e.g., object detection, monocular vision, drone autonomy, etc. While P is useful on its own for domain knowledge discovery, it can also be sampled by a generator to produce a dataset (D) to train and evaluate an AI model. However, in the real-world we do not always know what underlying information (I) drives T, let alone what D to collect in support of $\{I, T\}$. To date, we typically use expert knowledge to design a data collection under real-world constraints (time, money, etc.). However, this rarely translates into a dataset that is rich enough to train and/or truly evaluate a trustworthy, reliable, and unbiased ML model that operates across a range of contexts and environments. This trial-and-error process, which is how other challenges like material design work, is repeated until T is solved or funding is depleted. The point is, it is unlikely that a human knows I and will provide a sufficient initial P. This must be discovered iteratively in the real-world, in a surrogate environment like SIM, or most likely, a combined effort.

For sake of illustration, consider the following application. One of our research interests is object detection, identification, and dynamic interrogation of explosive hazards (EH) across environments, emplacements, sensors, and operating conditions (speed, standoff distance, etc.) using a low altitude unmanned aerial vehicle (UAV). For this particular application, task T is detection and localization of an EH. Information I that drives T can range from spatial shape features to multi-spectral, environment, platform, and/or emplacement context. We are

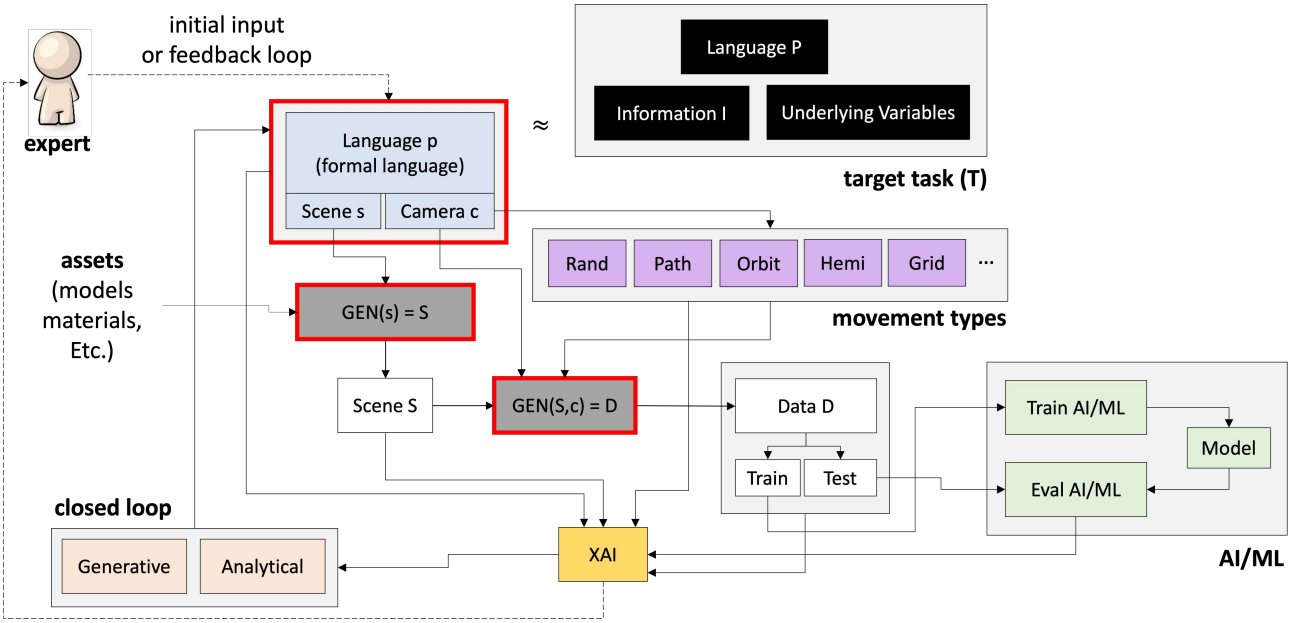

Figure 1: Our goal of a closed loop AI design and evaluation framework. Red indicates current article contributions. A target task is formally modeled via a procedural language. The resultant dataset is used to train an AI model and/or discover data, algorithm, and/or model biases and shortcomings.

interested in learning or discovering a language (formal task description) that is succinct yet sufficiently captures $\{I, T\}$. This challenge is extremely hard and costly to address in the real-world. SIM focused AI/ML has the potential to help us explore and address this task, or at least identify candidate solutions that can be explored further or in combination with the real-world in a more efficient and timely manner.

Herein, we put forth the following contributions. First, we investigate an explainable and scaleable procedural language for scene generation, called LSCENE, and data collection, LCAP. Next, we discuss data, ground truth, metadata, and details about how to produce, store, and format SIM information relative to state-of-the-art AI libraries, tools, and practices. Last, use cases and examples are provided to demonstrate the proposed ideas.

2. RELATED WORK

2.1 SIM Environments

A classical SIM environment, driven largely by robotics, is Gazebo.[1] While Gazebo is not state-of-the-art with respect to photorealism, it has extensively been integrated with the robotic operating system (ROS)[2] and it supports a wide range of physics and sensor SIM (e.g., RGB, RGBD, GPS, IMU, LiDAR, sonar, etc.). With respect to realism, Apple's HYPERSIM, Photorealistic Synthetic Dataset for Holistic Indoor Scene Understanding,[3] does not support physics but it produces nearly indistinguishable photorealistic imagery with information about scene geometry, material information, lighting, per-pixel instance segmentation's with respect to diffuse reflectance, diffuse illumination, and non-diffuse residual term that captures view-dependent lighting effects. Another approach is NVIDIA's Isaac Gym,[4] an end-to-end SIM platform, built from the ground up to run large-scale, physically accurate multi-sensor SIM for applications like autonomous vehicles and virtual worlds. The Unreal Engine (UE),[5] is another solution that has recently achieved near photorealistic results. UE has a generous licence that varies based on terms of use, ranging from free to percentage of profits. The UE started as a game engine and it has recently settled into new applications like architecture, film, computer graphics, and beyond. Last, a more recent competitor to UE is Unity.[6]

The above examples are from the robotics, computer graphics, gaming, and hobbyist communities. SIM dates back decades and its scope is beyond the current article. SIM frequently involves approaches like ray or finite

element modeling to accurately model topics like electromagnetics and physics. Examples include COMSOL[7] for multiphysics SIM and MEEP,[8] a free and open-source software package for electromagnetics simulation via the finite-difference time-domain (FDTD). These SIMs are often based on simulating sensors such as the visual spectrum (aka RGB) to multispectral, hyperspectral, radar, LiDAR, acoustics, and beyond. A recent physics-based example from the literature is the virtual autonomous navigation environment (VANE) and ANVEL.[9] The reader can refer to the internet for more large scale domain specific business examples like Scale AI.

While the number of SIM solutions seems endless, these tools are only a part of the big picture. How accurate are they? Most SIM solutions are provided "as is". That is, they have some level of trust associated with them, typically based on their design. However, most SIM solutions take shortcuts or they are discrete approximations to continuous processes. Only a few SIM environments have truly undergone a rigorous process of verification and validation (V&V). The reader can refer our article[10] for a recent historical review of the development of V&V theories for SIM models.

2.2 SIM and AI/ML

While Sub-Section 2.1 outlines different general SIM tools, the more relevant question to the current article is which of these have been used for AI/ML, and how? By far, the most work exists for training AI/ML/DL algorithms, e.g., synthetic data for DL,[11,12] imitation training,[13] selective instance switching,[14] indoor object detection,[15] driver assistance,[16] optical flow estimation,[17] urban semantic segmentation,[18] and adversarial training.[19] Work also exists in how to generate SIM ground truth, e.g., 5M photorealistic images of synthetic indoor trajectories,[20] playing for data,[21] and our recent article on flawed assumptions in SIM ground truth.[22] Another interesting focus is SIM digital twins and proxy worlds for AI/ML, e.g., virtual-KITTI.[23] While training is important, work also exists for evaluating AI/ML algorithms, e.g., evaluation of 3D reconstruction pipelines[24] and synthetic feature tracks for SfM evaluation.[25] Furthermore, while SIM is a wonderful place to experiment with AI/ML, works exist to try to bridge SIM models to the real world, e.g., domain randomization[26,27] and self-supervised monocular depth.[28] In many scenarios, we do not know what to SIM for AI/ML. Work exists to help learn what an AI/ML needs in SIM, e.g., autosimulate[29] and learning to simulate.[30] While full image SIM is just now starting to reach the point of photorealism where it might be able to be a good surrogate for the real world, a number of hybrid SIM and real data solutions exists, e.g., self-supervised image harmonization,[31] domain translation for image harmonization,[32] "cut, paste, and learn" for image synthesis and instance detection,[33] and a recent survey on deep image composition.[34] By no means is this a comprehensive list, these are just a few examples of SIM for AI/ML.

2.3 Procedural SIM

The process of generating content, e.g., textures, 3D models, etc., has been researched for several decades in the video game and film industries. In 1985, Perlin introduced algorithms around noise functions to create procedural materials.[35] Perlin first demonstrated these ideas on a marble vase. This procedural way of thinking grew to include methodologies like fractals and L-systems[36,37] for constructing plants, trees, and vegetation. Later works focused on real-time procedural terrains[38] and city generation.[39] While academically interesting, these ideas have been significantly scaled up in recent times by companies like SideFX in commercially available tools like Houdini. Many of these professional tools are now the standard for creating urban and rural cities in games and film. While much effort has gone into procedural algorithms for automatic content generation (including procedural creatures, animation, etc.), researchers have expanded this thinking and focused a users experience. For example, in Left4Dead, Valve introduced the AI Director,[40] a procedural algorithm that monitors user experience and changes the spawning and properties of entities in a map for dynamic pacing of game play. Procedural text-based games and procedural maps, e.g., dungeons, have existed for an even longer time in video games. The point is, while many procedural ideas have been put forth to date, most are focused on content generation and users (people). Relatively little research to date has gone into procedural algorithms for AI/ML. What type (volume, variety, and attributes) of data does AI/ML need? AI/ML is a relatively new "consumer" in the field of procedural.

2.4 Formal Languages and SIM

Successful video game procedural content generators still rely on the formulation of a grammar to specify the rules of generation. Not all grammars are created equally, however, and in the popular case of *No Man's Sky*, a video game developed in 2016, an L-system grammar was used for generation. This was done in an effort to better model natural systems that need to have high levelsXXX of details. Parallel rule rewriting in these systems help facilitate this process. As one branch of a simulated tree grows, each new branch grown will also be rewritten which leads to fractal like patterns. This process does allow for almost infinite detail, and is built from several discrete rules applied at each rewrite step. Simulating an entire level or entire scene, rather than a single object often requires a grammar with more control. Recently, DeepMind also presented an idea focused on AI called Open-Ended Learning Leads to Generally Capable Agents.[41] An agent played within simulated levels that had generic goals, both of which were procedurally generated using a formal grammar. These rules were capable of creating scenes such as "Put purple sphere near black pyramid" or "See the red player or stand on orange floor." These statements are then paired with four other statement descriptors which judge "Competitive, Balance, Options, and Exploration Difficulty." These metric functions take the difference between the maximum upper bounds of certain properties of scenes, and any new generated scene from the grammar sentence. As a result, a much more controlled environment is created for a reinforcement learning (RL) algorithm. Certain actions and other spatial relationships are present in this grammar, like holding, or being near another object. Huge amounts of meaningful data can now be simulated and generated procedurally, while still being able to maintain the appropriate level design to teach an RL algorithm.

3. PROCEDURAL LANGUAGES

This section describes the current state of our two procedural languages. While grammars and their implementations vary, solutions typically involve at a minimum *symbols* (terminals and non-terminals) and rules. To facilitate domain understanding, we desire a succinct language with understandable symbols and logic. Our language needs to also satisfy the request for mass content generation, while enforcing relevant rules. Furthermore, generated content should allow for strict labeling (markup) on every action taken. As a result of these constraints, we propose a rule-based procedural SIM language that relies on sampling a problem space, and an ordered hierarchy within a 3D scene. The next two sub-sections detail our current solution.

3.1 LSCENE: Scene Language

Table 1 is the combined LSCENE and LCAP symbol set and Table 2 is LSCENEs function set. We begin this sub-section with a working LSCENE example (see Example LSCENE 1). The aim is to help the reader develop intuition about what we have done before fine details are discussed.

Example LSCENE 1
Simple scene with a sky, single random object, and many rocks and bushes

```
LSCENE = ONE(S) ONE(W) ONE(O1) MANY(O2)
S = skybox{color=white,clouds:{ type={cumulus:0.5,nimbostratus:0.5},density=[0.1,0.2]}}
W = terrain{size=[201,201],height:{Perlin:0.2},material:grassy}
O1 = object{class:{A:0.5,B:0.5},rotation:{range:[0,0,0],[360,360,360]},position:{ON:W}}
IO1 = object{class:cube,visibility:false,extent[1000,1000,1]},position:{ON:W,AROUND:O1}}
O2 = object{class:{bush:0.2,rocks:0.8},scale:[1.0,1.5]},overlap:true,position:{ON:IO1}}
```

Example LSCENE 1 is a trivial environment that has a sky with a fixed color, one of two cloud types, and a range of possible cloud densities. Thus, this symbol (sky) has a combination of crisp, probabilities, and interval-valued data. The terrain is a fixed size, its geometry is determined by a Perlin noise function, and its material (texture) is of type grassy. Next, a single object will be placed on our terrain, from user defined class type A or B. The relative rotation and size of these objects have been specified, along with a spatial qualifier about where to place the object. Next, a non-renderable object is specified with a spatial qualifier. A number of other 3D objects, bushes or rocks, are placed in this volume. Overall, this simple LSCENE describes a grassy environment that has rocks, bushes, and a single object.

Table 1: LSCENE and LCAP Symbols

Symbols	**Meaning**
LSCENE	Start symbol for scene language
LCAP	Start symbol for capture language
S	Sky
DL	Direction Light
A	Atmosphere
SL	Skylight (Ambient Light)
W	Terrain
O	Object
CO	Camera orbit
P	Path
CL	Collection
SCS	Scene Change Set

Table 2: LSCENE Function Set

Function Name	**Meaning**
ON	Intersects a bounding box with the top of an object's bounding box
IN	Intersects a bounding box with an object's bounding box
AROUND	Intersects a bounding box with an object's bounding box with N-extent
ONE	Creates 1 object in LSCENE
MANY	Creates N objects in LSCENE
Position	Samples a 3D point from a bounding box and sets an object position
Rotation	Samples from a range of 3D rotation angles
Extent	Sets an object's model to extend to a given bounding box extent
Visible	True/false to display an Object's model
Bound	Sets an object's model extent to another object's size
Material	Samples a UE material and sets on an object's model
Scale	Samples between a min and max value to set an object's model scale
Overlap	True/false to allow an object's model to overlap other object models
Class	Adds a 3D model to an object
Blueprint Class	Inserts a created blueprint asset into an object
Perlin	Uses a terrain object and applies a Perlin noise function
Cloud	Uses a skybox and sets the specified cloud type.
Intensity	Uses a skybox and sets the lighting intensity (lux)
Density	Defines a percentage of a given input to use (clouds, atmospheric settings, etc)

Next, we proceed to a more detailed description of LSCENE. Table 1 outlines several components of a scene, namely, the sky (and all lighting), terrain, and any objects featured in this example scene. All symbols correspond to the bare minimum required to produce pseudo real SIM scenes. No constraints on the quantity of any element are specified, nor constraints on the type of each symbol outside of this minimal set (such as meta specifications like clutter, buildings, etc.). The goal of our initial language is to act as a basic interface for generating complex 3D scenes, not a specification that uses hundreds of properties about real objects. Additional rendering and scene details needed to achieve realism can still be used in tandem with our language. Each symbol can be arranged in an arbitrary hierarchical order. Many real world tasks require different constraints to hold true, such as geospatial imagery where atmospheric settings may override other lighting settings, or target classification which could need all targets to be fully visible. The JSON file is parsed sequentially, allowing back references to previously defined objects, such as lighting or terrain, which may be referenced by other objects. This simple format accurately covers what is involved in scene construction, and it allows for a minimum of rules to define a scene (easiest for user/algorithm readability and writing).

Our language relies on UE for 3D scene creation and manipulation. While our ideas can be extended to other SIM environments, UE was selected due to its diverse set of capabilities, photorealism, extended set of plugins (e.g., Infinite Studio for multi-spectral data), programming support, online community, licensing, and online free high quality 3D scanned assets like Quixel. Our LSCENE JSON file is translated into UE C++ functions and variables, and UE executes the instructions to the rendering pipeline. The world state is sequentially changed with each function. The overall speed of this system is constrained primarily by the initial JSON parsing. Afterwards, all code is in C++ function calls that maintain the typical speed of close to hardware programming languages. Our language aims to extend these UE functionalities, while preserving existing engine optimization and constraints. It also allows for flexible extension.

Functions in LSCENE only operate on the world state provided to them (the list of all current objects), and it simply uses the other inputs to derive the current context. For example, "Rotation(Object1, Rotation Properties, World State)" uses the object name to apply the rotation changes to the correct object in the world state. All functions in Table 2 make use of any built-in game engine functionality, such as creating, deleting, and manipulating object properties. The LSCENE functions are responsible for changing scene properties, thus fulfilling the goal of LSCENE to act as an interface for many scene constraints. All objects referenced by a function are evaluated in the order they are presented in the JSON, allowing the user to define priority. As an example, consider the object detection problem in computer vision, where target overlap and occlusion needs to be defined. Sometimes a variable amount of overlap, or closeness to other objects is desired. The size of a target can be made to be larger than normal, which will hide other objects. After the collision test is complete with the "Position" function, the real target size is applied, resulting in a gap between all other objects and the given target. The reverse function order can be applied to force a variable percentage of overlapping.

Although LSCENE provides plenty of functions to facilitate scene building, certain problems are not practically solvable with just the use of LSCENE. Terrain with complex geometry will result in less precise placement of objects. Content databases such as Quixel, Unreal Engine Marketplace, TurboSquid, etc., will often have incompatible materials or artifacts which are not accounted for when they were exported. Any asset with "errors" could result in LSENE producing incorrect scenes from the description given to it. Yet, even outside of these few examples where a user would need to first scan the content before blindly using, LSCENE has limitations. Creating scenes with any change over time will need to introduce more robust functions to account for the complexity. Thus, temporal elements and macro oriented structures within the language need to be implemented, which we define next via LCAP in Section 3.2.

3.2 LCAP: Capture Language

As in Section 3.1, we begin with an example. **Example LCAP** 1 produces a dataset with respect to a camera moving in a hemisphere (hemi) pattern around a target object of interest. First, CO defines the camera with a set of settings, field of view and output resolution. This camera exports target O1 with a per-pixel label of 5. Two post process effects (PPMaterial's) are added to the export data layers list (default data layers in Figure 12). The camera follows a hemi movement. A range of angles and positions are generated to capture multiple sides of an object. In this case, object O1 is centered about the origin, which is the default hemi orientation.

Example LCAP 1

Camera movement in a hemisphere (hemi) pattern around a target object to make a dataset

```
LCAP = ONE(CO) ONE(P) ONE(SCS) ONE(CL)
CO = camera{csettings{fov:30,res:[2048,2048]},targets:[O1:5],PPMaterial:[EdgeBlur,CartoonStlye]}
P1 = path{create hemi:{cam radius:[1000,2000,5],obj angle:[0,360,4],cam angle:[45,45,1]}}
CL1 = collection{C1:{move:[P1:3]},O1:{when:{after:P1:1},material:MetalGold},{when:{after;P1:2},
      rotation:{range:[0,0,0],[100,100,0]}}}
```

Finally, the collection is defined, which defines the behavior of our scene over time. Symbol "move" generates a list of transforms on our camera. The notation "P1:3" generates 3 copies of the path defined with labels P1(1), P1(2), and P1(3). Symbol "when" tells when in time to apply a scene change. This function takes in a reference to an instanced path, e.g., P1(1), and triggers after the first iteration of P1 is complete, applying a different material to O1. This function can be repeated any number of times, and also triggers after P1 iteration 2 is complete to apply a random rotation to O1. Any and all LSCENE functions are callable here.

In general, LSCENE specifies a static scene. Many applications, however, require more parameter changes, e.g., scale variations, material changes, or slight positional perturbations over time. The capabilities of LSCENE therefore need to be extended. LCAP was created to solve this dilemma. Furthermore, collecting information from within a 3D scene has specifications outside of the physical and material properties of a scene. Tying the SIM scenes to their real counterparts highlights another important concept LSCENE will not account for: platform operating conditions. How is LSCENE being sampled? What are the settings on the camera doing the scene capture? What platform positions and viewing angles are needed? Accounting for these added complications results in LCAP being a higher-level language that leverages the existing functionality of LSCENE to cover a wide range of data collection requirements.

To reiterate, LSCENE lays the groundwork for manipulating data in a scene. While LCAP is responsible for changing the scene with time, setting the platform operating conditions, and managing which data is ultimately exported. Section 5 represents some of the minimum use case requirements of LCAP, e.g., scanning a single object, grid and random search, etc. More so, AI relies on large amounts of labeled data which LCAP should facilitate through several of the techniques discussed in Section 4.2, like domain randomization. These ideas (data augmentation, exportation, and variation), are some of the motivations for LCAP. Every function in Table 3 leverages object transforms and data types defined in LSCENE. Just as LSCENE is using UE functions to manipulate scene objects, LCAP is using LSCENE for the same purpose. This uses the capability already developed in LSCENE.

Another LCAP complexity is how it uses LSCENE data types to apply changes to other objects. Every time an object is referenced in a collection, LCAP calls an LSCENE function with a specified input. Even camera paths are created via this method of passing LSCENE functions into LCAP functions. First, a generic point (an object) with a 3D box representing noise in space is created. The same LSCENE function calls used for any other object with these details are used. Essentially, this acts as a blueprint for LCAP to use and generate more points. Then, LCAP uses an "increment point" function to generate N points in a grid like structure across another specified 3D space, in a similar fashion to other functions that reference objects in LSCENE. The end result is a uniform grid over a terrain (or object) with variable perturbations in the x, y and z dimensions. This is desirable, as operating conditions in many problems have noise in the real data, so it is pivotal that LSCENE/LCAP can introduce noise as well.

The concept of sets in LCAP should be discussed. These sets are composed of LSCENE objects with some number of relevant properties. In the context of "move", a path (which is also a set), is taken in and the position and rotation properties are used to generate the respective "position" and "rotation" LSCENE functions on the camera, or object. The last LCAP function that needs significant explanation is "when." Video editing tools rely on key-frames to apply some transformation at a specific time. LCAP can use frame numbers to apply changes, but this is often not desired. Rather, by naming collections and creating many instances of them, a user is able to create minimum sized blocks of time in which changes take place. References to these sets currently rely on temporal terms like "after" and "during." After will produce changes once a collection finishes, right

Table 3: LCAP Function Set

Function Name	Meaning
Target	Set the semantic segmentation ID of a given object
Camera Settings	Sets the camera FOV, image output, focal distance, and focal length.
Capture Settings	Configures UE render location output and final scene collection folders
Create Point	Defines a 3D rotation and 3D position sampled from a 3D position/rotation space
Create Hemi	Uses camera radius, object angle, and camera angle to create 3D points
Increment Points	Creates 3D points in a 3D space, given some separating extent (x,y,z)
Create Path	Defines a space and a number of points to generate within the space
Increment Point	Uses an x,y,z offset amount, to create the 3D positions of points in a path
Scene Change	Generates a set of scene transforms on an object (material, scale, etc)
Create Set	Concatenated scene changes and paths into one labeled collection
When	"When" in time to apply some scene change. Uses a set, path, or frame number.
During	During every frame tick on a set, some scene change will be applied.
After	When a set, frame number, or path is finished, apply some scene change.
PP Material	Apply a post process material to the render output

before the next collect starts. During applies a scene change every moment in time in a collection. These are not all encompassing terms, rather they are starting points for many applications, in the same way the spatial constraints proposed in LSCENE are starting points for many spatial relationships.

A more abstract language is proposed in LCAP. Once a 3D scene is expanded in the time dimension, so must the language. LCAP defines a robust collection reference system that reduces unneeded specifications when timing scene changes. Set combinations, complex way point generation functions, and data export settings all reside in LCAP. As more functions are added to LSCENE, the capabilities naturally grow in LCAP. This is the core connection of LCAP and LSCENE. Even if LCAP adds more functions, this will not change the fundamentals in LSCENE. Therefore, we have created a formal system that relies on a set of symbols, see Table 1, functions (operators), see Tables 2 and 3, and the ordering of these symbols and operators to produce meaningful 3D scenes with properly labelled data.

4. WORKING EXAMPLES

To fully appreciate LSCENE/LCAP, one must see it in action. The next sub-sections highlight two simple scenes.

4.1 Example 1: Simple Biome with a Single Object

Figure 2 shows nine random images from a data collection that could be used to train and/or evaluate (in conjunction with produced truth) an object detection and localization algorithm. Specifically, Figure 2 shows photorealistic SIM imagery for a simple scene with *a few* target objects (blue water cans) in an open grassland biome. This LSCENE specified *some* foliage objects (bushes) on a grassy terrain (different textures/materials) with sparse grass. The corresponding LCAP specified an interval of off-Nadir drone poses, operating altitudes, and times of day. A large part of this simple LSCENE example is foliage. Figure 3 is an example of the foliage section from our LSCENE JSON configuration file.

4.2 Example 2: Randomized Clutter

Section 4.1 focused on one of the most trivial scene and data collection scenarios possible, a single object type, terrain, and vegetation. In example 2, we show how a minor change to LSCENE and LCAP results in a great deal of complexity that could be used to train or stress test an AI algorithm. In Figure 4, we changed the terrain texture/material and bushes to obtain a rocky sandy biome. Basic regex statements allow for precise asset discovery from a local content database or an API like Quixel. The decision to change the biome was just so the reader could see that LSCENE maintains efficacy across different environments. Next, we specified a *massive* amount of random clutter in LSCENE. This is just a change in the range of possible objects and pointing LSCENE to a clutter folder to sample from. The reader should note that this scene is not designed with realism

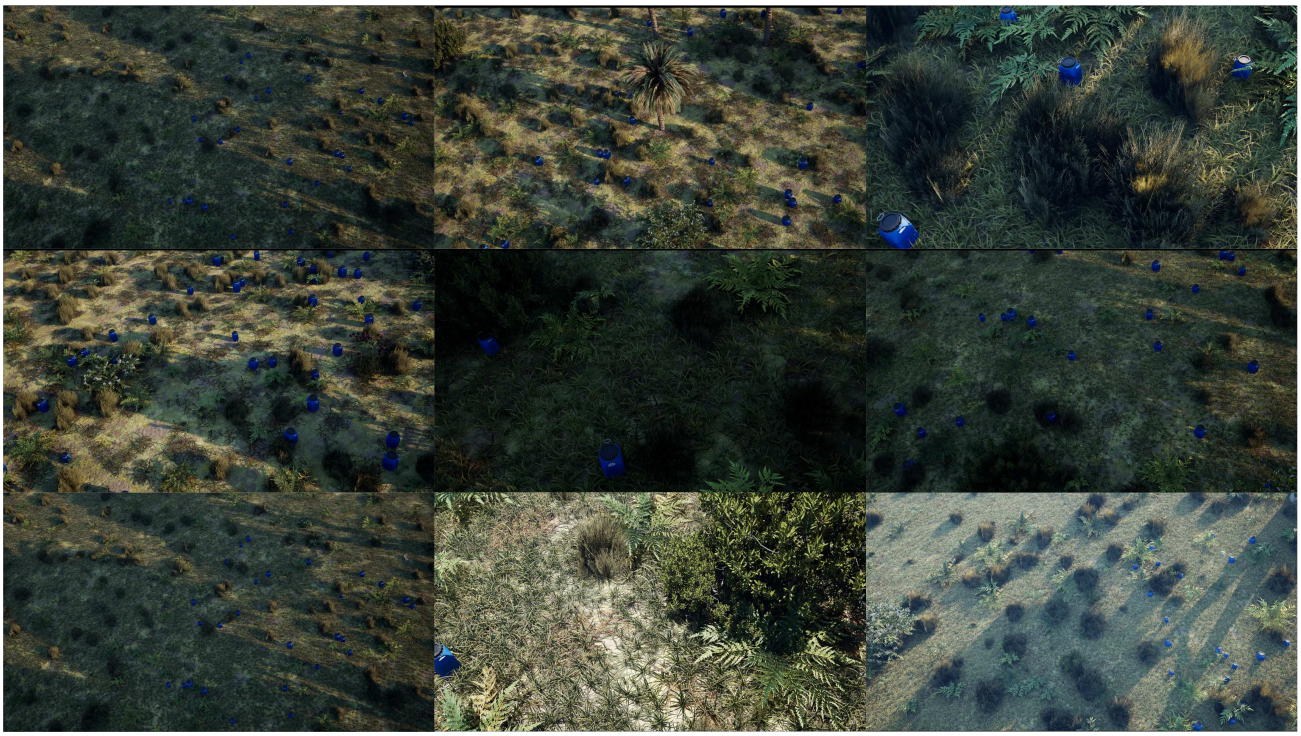

Figure 2: LSCENE Example 1 for a grassland biome and target object. See Section 4.1 for more details.

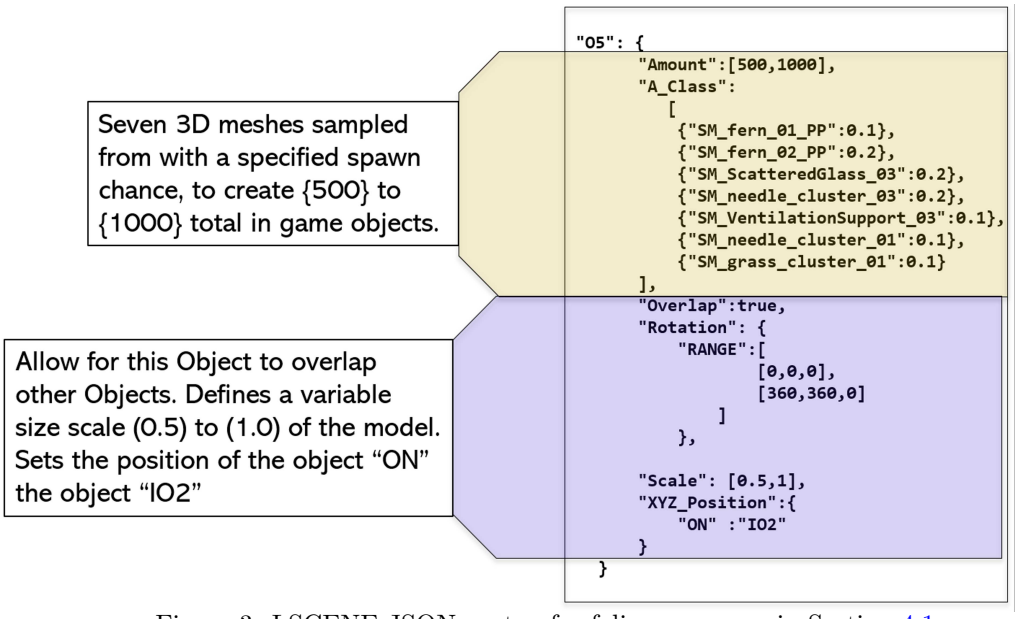

Figure 3: LSCENE JSON syntax for foliage spawner in Section 4.1.

in mind. The aim is to sample a range of clutter, nature and man made, with varying sizes, textures, colors, and other attributes. Current generation AI/ML algorithms require a great deal of data, and variety at that, to learn and ultimately generalize. A challenge is, if we generate only a single scene, we can collect a dataset. However, we would be getting a large number of looks on a relatively few number objects. This is a similar situation, and potential problem, of most real world data collections. Instead, LCAP acts like a modifier to LSCENE and in each frame, objects textures/materials and other user specified attributes, e.g., size, rotation, etc., are

Figure 4: LSCENE/LCAP Example 2 for scene with random manmade clutter. See Section 4.2 for more details.

Figure 5: LSCENE/LCAP Example 2 with randomized natural clutter. See Section 4.2 for more details.

varied. The aim is to realize a reduced size dataset with high inner scene variation via randomized sampling. AI algorithms do not always scale well with data and the *cost* (computation, storage, financial, etc.) required to train and evaluate models on Big Data can be a serious, if not limiting, problem. LCAP was designed with these constraints in mind. These ideas are partially motivated by recent findings in related topics like domain randomization and sim-to-real.[26,42–45]

While Figure 4.1 is an example with suspended realism, Figure 5 is a realistic and natural looking scene. Namely, in Figure 5 we do not insert tons of arbitrary objects (tires, boxes, barrels, etc.). Instead, assets that are compatible with that biome are specified and sampled. The result is a more realistic nature (vs man made) cluttered scene, but proper content labeling is required to properly grab biome specific clutter. Quixel (a 3D content delivery service) supports a tag based labeling scheme, with object identifiers such as desert, plants, arid, rocky, etc. This does allow for procedural asset selection as it pertains to biome compatibility.

In closing, some applications, e.g., where context does not perhaps matter, stand to benefit from extreme man made clutter insertion and randomization (Figure 4.1). However, for applications where context is important, a LSCENE like Figure 5 can be used to specify complex natural scenes and more fine detailed LSCENE language features like spatial modifiers and allowable occlusion percentage can be used to control where objects are placed and how those objects are viewed by our camera/platform. Last, another set of applications could benefit from complex environment realism and randomization, e.g., Figure 5, coupled with a limited amount of randomized man made clutter insertion. AI algorithms are not just about learning a target class. The machine must not only learn features and logic to recognize desired objects, but also features and logic that can discriminate those objects from other confusers. It remains an open research question if one of these strategies is superior and applicable to all applications. Instead, it is more likely the case that different challenges require different approaches. This is how we designed LSCENE/LCAP. It is not a single tool for a single approach. Instead, it is a single approach that can do many different things and we ultimately leave it up to the AI to help us decide

what strategy is the most effective.

5. EXAMPLE AI/ML USE CASES

The last few sections focused on simple and analytically tractable LSCENE/LCAP examples. In this section, more complex scenarios from our computer vision and drone autonomy research are provided. These use cases (UC) highlight applications of the proposed tools and processes to date.

5.1 UC1: Object-Centric

Use Case 1 (UC1) is focused on generating SIM data in support of a target object. Two examples from our group include generating then inserting SIM objects into real imagery (UC1a) and full image SIM for evaluating a trained AI/ML model (UC1b). In UC1a, we previously introduced a method called altitude modulated augmentation (AMA),[46] which intelligently inserts a hemisphere (hemi) or orbit collected SIM data into real drone imagery with respect to metadata to directly train or augment a DL model in domains with low-to-no data. Figure 6 highlights the scripted movement of a camera and sun around a stationary object with varying attributes (color and texture). This is a trivial LSCENE/LCAP program. LSCENE has a single terrain, object, sky, and sun. LCAP has a direct function built in (see Table 3), Create Hemi, versus requiring the user to calculate and specify the required set of waypoints and relative poses for this movement type. Figure 7 is a full SIM output image (person on a road) and a white benign background template, which AMA uses to insert our object (person) into real imagery. In scenario UC1b, LSCENE/LCAP can be used to make a full SIM dataset in different contexts (environments, occlusion levels, emplacement contexts, etc.) for explainable AI (XAI). In,[47] we performed a camera-object relative scan to mimic a low altitude drone moving around a point of interest (potential target/object). This dataset was then subjected to a DL object detector built using real world data. From here, we produce a set of graphical and linguistic XAI summarizations that highlight our data, algorithm, model successes, and shortcomings.

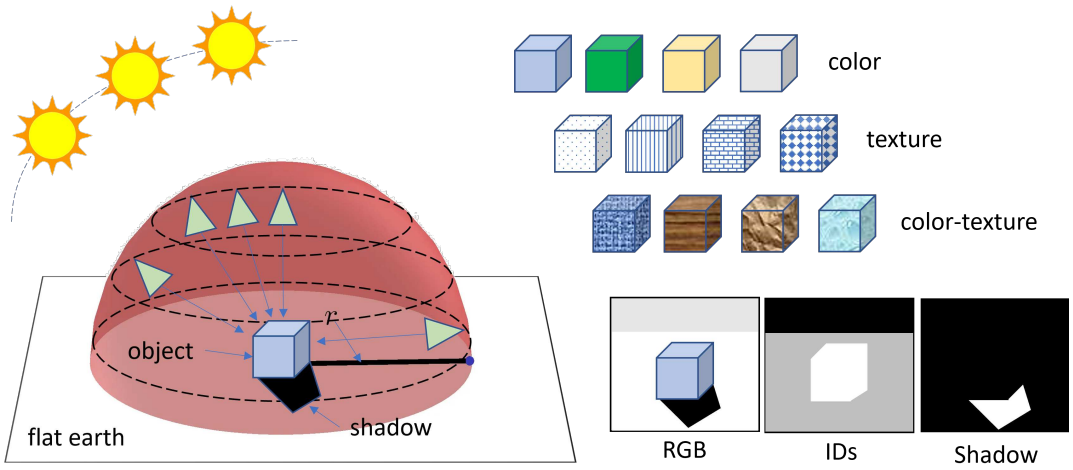

Figure 6: UC1: Hemisphere scan with corresponding data layers.

5.2 UC2: Randomization

In UC2 (Figure 8), LCAP can be used to generate a randomly sampled dataset for a desired operating context. For example, we used this philosophy to build a dataset to train and evaluate data-driven monocular vision algorithms across environments and relative viewing contexts.[48] While a grid or pre-determined waypoints can be useful, it is a pattern and therefore subject to potential bias. Grids and waypoints are also not efficient strategies for sampling large combinatorial object, environment, and platform variable spaces. In the literature, it is well documented (but not proven) that variety is equally, if perhaps not more important, than volume for training AI. When the variables that drive a task are not well understood, or when no specific experiment has

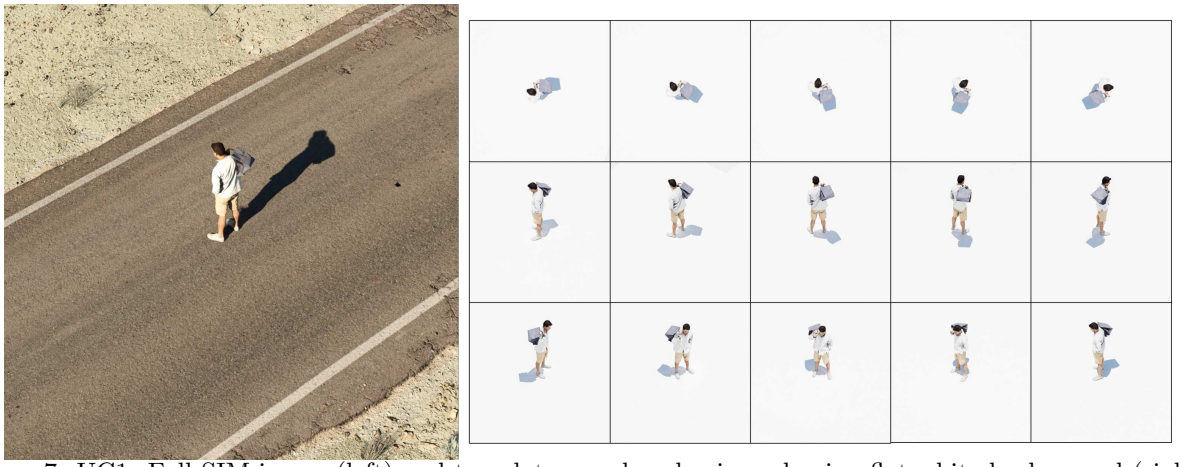

Figure 7: UC1: Full SIM image (left) and templates produced using a benign flat white background (right).

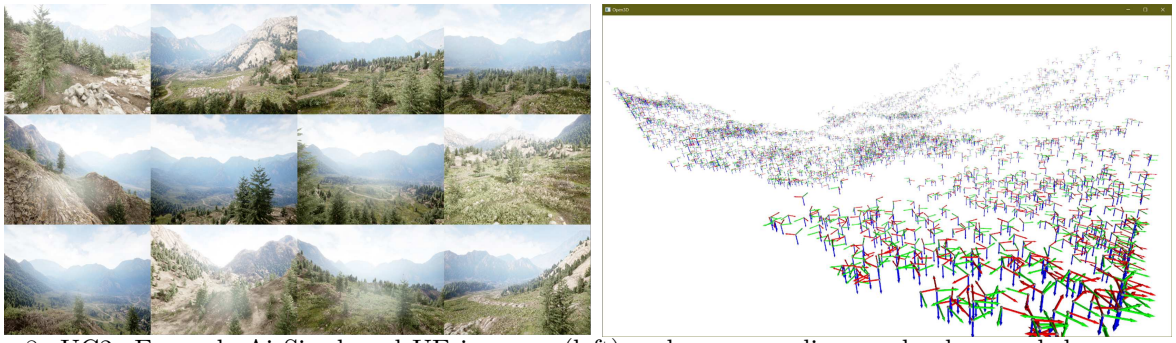

Figure 8: UC2: Example AirSim-based UE imagery (left) and corresponding randomly sampled camera poses (right) around a user specified drone operating context (desired range of altitudes and poses).

been outlined, random data collection and analysis, over repeated trials can be an effective strategy for training and helping understand a given AI. These are just a few of the many reasons why we built a random move capability into LCAP.

5.3 UC3: Motion-Centric

In Sub-Section 5.2, we argued for inclusion of randomization in LCAP. In this sub-section, we argue the opposite, inclusion of grid and custom waypoint motions. Figure 9 shows example controlled LCAP movements in support of structure from motion (SfM) algorithms. Recently, we introduced a SIM framework and workflows to help understand and characterize hand crafted and data-driven ML SfM algorithms.[49] We used the various data layers, truth, and metadata from SIM for evaluation at each image and across images to build a gold standard and ultimately focused metrics that compare and help us select algorithm parameters and ideal flight contexts (altitude, camera pose, etc.). We also used this LCAP capability to capture uncertainty in monocular depth estimation by constructing fuzzy voxel maps.[50] Figure 10 shows how this process was used to improve the quality (accuracy of free, occupied, and unknown space) of an aggregated (multi-look) voxel space for SfM. Last, we also used this LCAP capability to build datasets to train and evaluate a data-driven monocular vision algorithm for passive ranging in the context of autonomous ground vehicles11. In summary, there are many AI, computer vision, and drone autonomy scenarios that required controlled camera movements.

6. DATA, GROUND TRUTH, METADATA, AND FILE FORMATS

The previous sections focused on tools and examples. This section details specifics regarding data needed by our algorithms during training and/or evaluation. Our focus is breadth and coverage vs a single AI workflow, e.g., what's the optimal way to generate and train a detection and localization algorithm like You Only Look Once (YOLO).[51–53] Hereafter, instead of calling everything *data*, the following terminology is adopted.

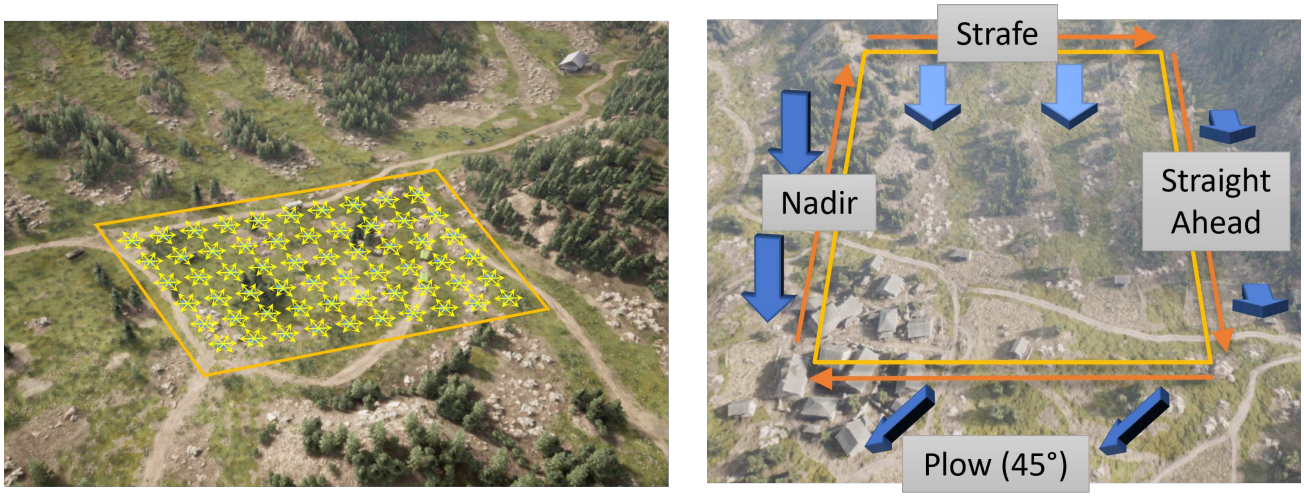

Figure 9: UC3: (left) Grid collection and (right) custom waypoints with different viewing conditions.

Figure 10: UC3. (right) SfM reconstruction on SIM data relative to (left) its corresponding gold standard.

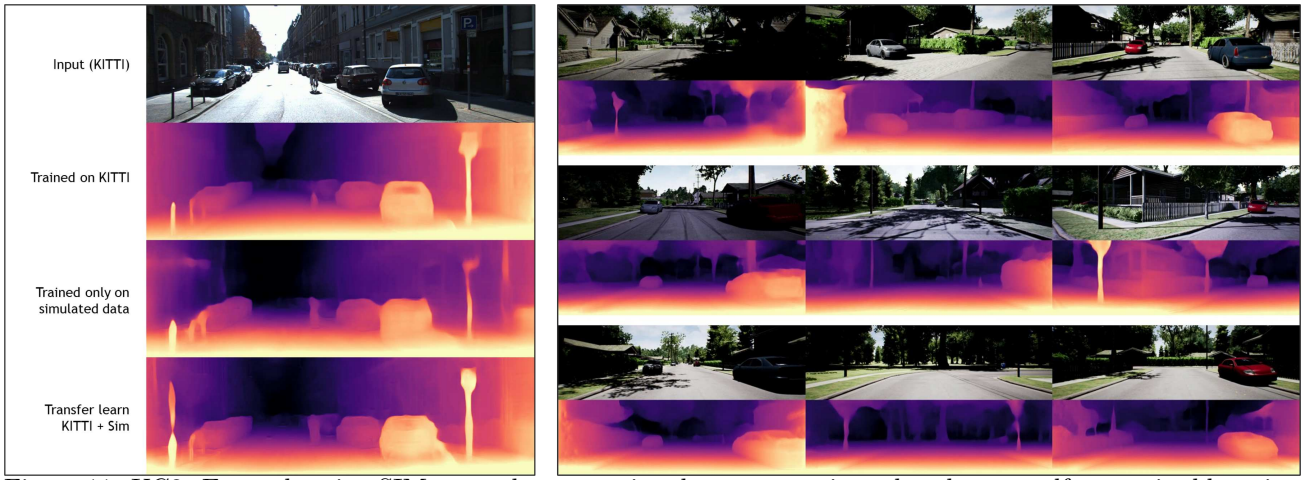

Figure 11: UC3. Example using SIM to produce a motion dataset to train and evaluate a self-supervised learning based monocular vision DL algorithm. (right) Example SIM training data with associated depth truth. (left) Example of a real, full SIM, and hybrid models evaluated on KITTI vehicle dataset.

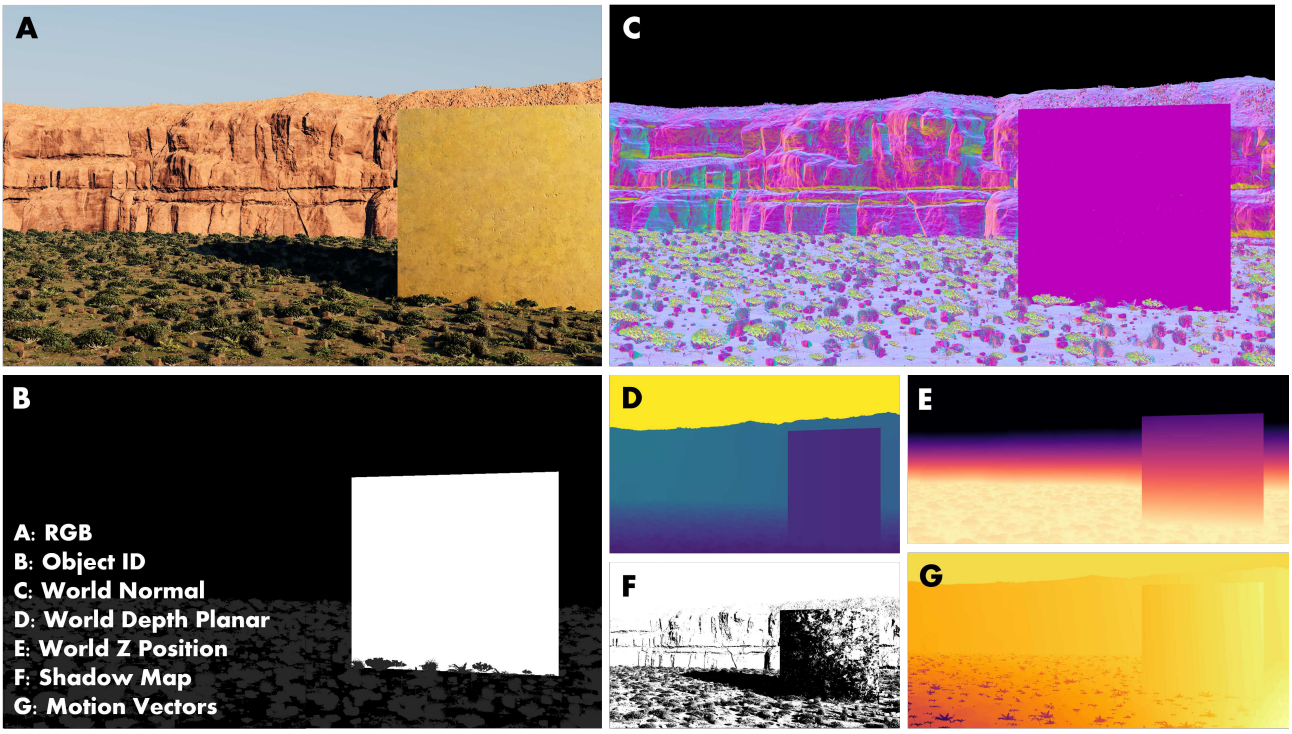

Figure 12: Example data and ground truth layers. See article for additional details.

- **data**: this is typically raw or normalized numeric values from a passive or active sensor that the AI/ML is processing and using for decision making. Examples include imagery from a visual spectrum (aka RGB) camera, raw 14 bit normalized (e.g., mean absolute difference (MAD)) infrared imagery, Velodyne LiDAR puck position and/or magnitude values, etc.
- **ground truth**: information about the objects, environment, events, context, and other factors that define and drive our task. Unlike data, this information is not explicitly available to an AI at test/run time. Examples include object identifiers (e.g., class labels), bounding volumes (e.g., maxis aligned bounding boxes (AABBs)), emplacement photos (additional imagery that helps a user better visually understand the scene and/or objects), relative or global reference frames and coordinate systems (UTM, WGS84, MGRS, etc.), depth (relative or global reference frame), 3D surface/point cloud/voxel space, surface normal's, object positions and poses, temperature, time of day, etc. In the real-world, most ground truth is assumed fixed/static in time. However, simulators have the potential of storing such information at the expense of disk space and complexity at each moment in time (frame) if/when desired.
- **metadata**: numeric values measured at usually different sampling rates by the platform and sensors. Examples include position, pose (roll, pitch, yaw), temperature, light sensor that measures irradiance, camera settings like white balance, gain, etc. As metadata is measured, it has associated error.

As the reader can see, these terms do not have clear boundaries. Herein, data is the primary input (e.g., image) to our algorithms, ground truth is not available at run time, and metadata is measured data that helps an AI/ML better understand and/or process its data. The following sub sections detail our data generation, layers, file formats, and organization. Figure 12 are examples of what data we generate and store.

6.1 Data Layers

This section focuses on information that an AI has and can use for online decision making. Specifically, we restrict the focus of our current article to RGB imagery. Figure 13 is an example of the photorealism possible

with UE5. If the reader wishes to simulate multispectral data in UE, then the Infinite Studio plugin can be used[54*]. First, there is no single accepted way of generating RGB SIM data. Example approaches involve traditional and efficient rasterization, ray tracing, FDTD, etc. Furthermore, each of these are approximations and what parameters should we use? This is a challenge, all this freedom, in part because we still do not really understand what "type" of data AI/ML needs. Does the RGB imagery need to be 100% accurate? What level of realism is acceptable? Furthermore, are there any "fingerprints", things present in SIM that are not in real data, that a machine might pick up on, create unwanted biases, impact our SIM-vs-real world discoveries, or ultimately, prohibit our ability to transfer or use SIM data/models to the real world?

Figure 13: Example RGB urban and rural photorealistic images from UE5.

The following is our current RGB workflow. AirSim[56] was not used because while it is useful for simulation of an aerial platform, it results in less than photorealistic RGB imagery. Instead, we use UEs Movie Render Queue (MRQ), which is used for offline non-gaming processes like film, for rendering with deferred passes to make RGB and other layers. We developed a C++ Plugin and talk to the UE MRQ library. RGB imagery is stored in the EXR file format. EXR is an open-source high-dynamic-range image file format created by Industrial Light and Magic. Specifically, we use OpenEXR. Beyond storing higher bit depths (beyond 8) and addressing compression (e.g., uncompressed), OpenEXR allows for arbitrary and custom channels like specular, diffuse, alpha, normals, and various other types like we discussed in the ground truth sub-section. OpenEXR also allows for storage of sensor or rendering metadata, which is often needed to understand the generation and/or address calibration. Our approach involves letting the producer (e.g., the MRQ) generate its EXR data however it deems fit. At a minimum, we require red, green, and blue layers and fields specifying the width, height, and number of channels. In addition to EXR RGB imagery, a JSON is produced with custom description of how the data was produced. We discovered that many of our documented details were not known ahead of time. This is one part an evolving set of UE tools with frequent updates, and another part the fundamental nature of open research[†]. For example, a user might specify in our JSON that the UE5.2 engine was used with Lumen, along with any other relevant miscellaneous markups they see fit (e.g., lux settings, volumetric fog, screen space ambient occlusion settings, etc.). In the majority of our current AI/ML workflows, we use the following settings. We disable per image antialiasing (AA), disable temporal multisample effects, enable spatial multisampling, and Game Overrides[‡].

6.2 Metadata

Herein, metadata is information observed at runtime by the platform and/or camera. Ground truth records real (true) values. We store metadata in a JSON and they are measured observations. If an engine and simulation is perfect, then metadata and ground truth are equal in our approach. However, libraries like AirSim exist to model

*Simulating multispectral data does come at an extra added expense of extended material and physics specification and simulation, which can add significantly more processing and offline content modeling time and expense. In comparison, free resources like Quixel[55] exist for high quality real world scanned RGB materials and 3D models.

†Albert Einstein's famous quote, "If we knew what it was we were doing, it would not be called research."

‡This include Cinematic Quality Settings, texture streaming, level of detail (LOD) zero, disable hierarchical LOD, high quality shadows, overide distance scale, and extend view distance scale

drones and real-world flight. Furthermore, some SIM investigations intentionally introduce controlled error at different levels to assess its impact on AI. In any respect, metadata is simply addressed herein as a single JSON file with true or perturbed measurements.

6.3 Organization

To accommodate a range of end users and AI applications, we adopted the following strategy. Each data and truth layer is stored in as an EXR file in a separate folder with a simple name, e.g., "rgb", "worlddepth", etc. After attempting a number of complicated strategies, we discovered that less is more. More sophisticated ideas unfortunately required frequent changes or they conflicted with end users and their niche applications. Our flat organization worked best and the accompanying JSON files allow for customization. Overall, we strive for generality and minimal user constraints. Any consumer, e.g., an object detection and localization algorithm, can easily parse this flat structure, look for keywords in the folders like "rgb", and parse the JSON to determine attributes and if they want to use it or not for training and/or evaluation.

6.4 Ground Truth Layers

This sub-section is perhaps one of the most compelling reason to couple AI and SIM. Its incredibly hard, if not impossible, to get accurate truth in the real world. The act of having and exposing truth to AI is a game changer. Truth can be used in a number of ways from simply scaling up AI (making more data) to improved supervised learning (vs resorting to unsupervised, self-supervised, or other paradigms) and deep contextual evaluation. In anticipation of needing data sets for various use cases, we always output the following (see Figure 12) layers: object IDs, depth, shadows, surface normals, motion vectors, and world locations.

Object IDs: Object IDs are a per-pixel class or instance coded layer. Typically, this is a monochromatic integer valued image where the closest object ID is recorded[§]. Modern engines like UE typically tackle object IDs in an efficient fashion via a stencil buffer, which allows for $2^8 = 256$ unique IDs[¶]. The reader can manually extend the number of supported IDs by using a strategy like a custom integer or floating point render buffer with an ad hoc encoding. While 256 IDs may be sufficient for environments with a small number of object classes, dense and/or complex modern scenes like Epic's photo realistic Matrix City has hundreds of classes and thousands to millions of object instances. We do not store instance IDs by default as the resultant floating point-based file sizes add up quickly and most workflows do not take advantage of instance IDs. The reader should also note that a large percentage of pixels in an image are "mixed", i.e., the volume in space covered by a pixel includes multiple objects. Although this is true, its of relatively low value to a game engine. As such, engines only store a single or mixed (e.g., average in the case of anti-aliasing or spatial or temporal up sampling) value versus a list of all objects[‖]. In summary, we currently store integer-valued object class IDs with a single (not mixed) per-pixel value. We have not found nor logically been able to identify an AI/ML application that welcomes averaged object IDs. Such an operation results in IDs that belong to another class[**].

Depth: We record depth in world space units. Many engines are capable of producing different types of depth. For example, UE supports planar (all points that are plane-parallel to the camera have same depth) and perspective (depth from camera using a projection ray that hits that pixel). In UE, depth is recorded in cm or meters,

[§]The UE MRQ supports exporting object IDs to the EXR file format.[57] The reader can pick full, material, actor, actor with hierarchy, folder or layer. For example, selecting actor tracks the location of each pixel for each actor in the image. While useful, this can result in extremely large file sizes. For example, an image with resolution 2,048 squared results in approximately a one terabyte EXR file.

[¶]We used deferred rendering and included depth into UEs post-processing stencil buffer material to address occlusion. Otherwise, IDs of occluded pixels are recorded in the ID map (see Figure 12). This is sometimes a debated topic. For example, consider the case of a car occluded by a tree. Should an object ID map include each pixel for the car, occluded or not? This is very application dependent. We stick with the typical convention of recording just closest non occluded object IDs. If the reader wants all pixels, occluded and not, then the MRQ and its object ID exporting in EXR format can be used versus a stencil buffer.

[‖]See our 2023 SPIE paper about what is a pixel, does simulation really have truth, and how can biases be mitigated in training and evaluation via recording and using multiple samples.[22]

[**]For example, let object 1 be ID 10 and object 2 be ID 2 and assume a mixed pixel with half object 1 and half object 2. The averaged object ID would be 6, which is not object 1 or 2.

depending on how the user configured their project. Figure 12 shows an example depth map, which has been color coded between min and max value for display purposes. Details of interest include the following. We store depth as a double for precision. A single depth value is recorded per pixel, versus a combined (e.g., averaged) value. If mixed pixel depths are averaged then "3d point cloud trails" result. This is unacceptable for the AI/ML applications we have explored. This is why we use a single depth from one of the objects vs an incorrectly assigned number somewhere in free space between the objects. For example, consider a close tree and mountains at a far off distance. Depth pixels on the edge of the tree would have averaged points at the half way point of the mountain and tree. The reader can see our 2023 SPIE article for how to improve this process by recording and using a pixel bundle for unbiased training and evaluation.[22] However, such a solution is a tradeoff of accuracy versus storage and compute.

Shadow: Shadows are often discussed in terms of umbra, penumbra and antumbra. However, this information is not of general use to game engines and it can be expensive, is possible at all, to extract. In our experience, users are typically interested in relatively simple shadow information, e.g., a binary indicator if an object is in shadow or not (independent of a particular light source). As such, we have adopted a computationally efficient and extremely simple strategy where the difference, or a ratio, is generated between the base render pass and the final rendered image. Figure 12 shows an example. The result is a single floating point shadow image, where 0 is full shadow and 1 is no shadow. It is important to note that this method has limitations. For example, dark (e.g., black) surfaces will obviously not be addressed appropriately. Furthermore, the intensity of lighting is not well accounted for, e.g., this value changes with respect to lux. This is also an inclusive calculation, e.g., it considers rendering approximations like ambient occlusion mapping and parallax occlusion mapping. While simple, this has been useful in our research with respect to simulating objects and inserting them into real drone imagery.[46] We scan an object on a white background. Pixels on the ground, identified using object IDs, that are darkened, i.e., are not full white, are locations where our object is casting a shadow (on flat earth). If detailed shadow information is important to the reader, a strategy beyond what we have outlined here is necessary.

Normals: Many applications are concerned with estimating and using 3D information like points, voxels, and/or surfaces. Offline, normals have use in tasks like evaluating and understanding 3D AI/ML algorithms, e.g., SfM, MVS, etc. In real time, a number of recent ML/DL models have been proposed to estimate normals from a single image for tasks like object pose estimation and/or detection. Regardless, herein we store unit length surface normals in the engines world reference space[††]. Figure 12 shows an example three channel world space normal image, where red encodes x, green is y, and blue is z. As discussed in the previous sections, we store a single normal per pixel, not an averaged value. While an averaged normal along an object seem appropriate, averaged normals along the edges of objects are not. Conversely, a bundle can be computed and stored.[22]

Motion: In many applications, e.g., object detection, tracking, and depth estimation, optical flow and motion estimation is important, with respect to training and real-time decision making. Herein, we make use of the UE motion vector deferred rendering pass.[57] Specifically, motion vectors are stored per pixel in $[0, 1]$ for x and y, where 0.5, 0.5 is no motion. UE stores motion vectors normalized to the entire screen.

JSON: The past few paragraphs focused on layers as images. Our group produces a single JSON per data collection. This JSON has an entry for each image and we store information like 3D OBBs, 2D screen space AABBs, time, position, and orientation per object of interest. This information is crucial for algorithms like object detection and localization. Specifically, we store this data in a simple JSON file because there are various specific file formats used in practice to train and score algorithms. For example, YOLO uses Darknet TXT and Computer Vision Annotation Tool (CVAT) has its own file format for labeling and viewing results. The point is, we store a single simple JSON and write translators to whatever utility it needs to interface with.

7. CONCLUSION AND FUTURE WORK

Modern AI is unfortunately heavily dependent on large quantities of high quality labeled data. In this article, we discussed our design and implementation of two procedural languages, LSCENE and LCAP, for synthetic data generation in the context of AI and low altitude aerial drone applications. Analytically tractable examples

[††]The default units in UE are cm, positive y is right, positive x is forward, and positive z is up.

were given and more complicated use cases were discussed. We also detailed our data, ground truth, metadata, formats, and storage processes.

In future work, implementations of more robust LSCENE and LCAP functions will be explored. Currently, the spatial constraints in LSCENE have a precision only up to a 3D bounding box. Extending the core functionality of each function presented in LSCENE/LCAP, will hopefully result in more transparent and intricate scenes that better reflect the description of the language. Once the overall competency of LSCENE/LCAP is finalized, an AI/ML algorithm would be able to generate scenes automatically. For every scene auto generated, a neural network, genetic algorithms, or other learning method could help produce scenes that better reflect a desired goal. These goals can vary in nature, but a basic example would be for target detection, where data is produced to minimize a classifiers error rate. A formalized logic system in tandem with some system that builds truth statements from a SIM scene, could also find use in building and describing complex scenes. UE has a header tool which automatically creates a reflection system of all of the functions and types in their engine. A formal system can then query this reflection tool about possible functions, query for the parameters and parameter types, and then call these found functions with good inputs that lead to the same minimization problem as before. Our LSCENE/LCAP framework already reduces the search space a great degree, but to advance the language more, fundamental questions about the engine are asked through this reflection system to introduce new terms into the more abstract LSCENE/LCAP language.

Last, the current article is primarily documentation and an open discussion about how to formally describe, sample, and produce SIM data in support of AI/ML. In future work, we will take the next step and explore automated closed loop language learning for a specific real world task like EHD. While two objectives are the efficient production of SIM data at scale and training high quality AI/ML models, another real-world objective is opening up hood and analyzing the learned language (variables and rules) for domain knowledge discovery. Before this feat can be accomplished, however, we must first research and develop the specification and production steps outlined in this article.

APPENDIX A.

In this appendix, we summarize two algorithms used by LSCENE and LCAP. Any function in Table 2 has the potential to be probabilistic or deterministic. Inherent to each function, however, is the nature of sampling from many possible solutions. Essentially, every function is called with some lower and higher bound for what the function output can be. In this appendix, we can examine Algorithm 1 to understand how each function "space" is confined or expanded. Any function which can take in a range of inputs, will sample the inputs in the same manner. First, 0 to N inputs are provided for a function. Next, f(inputs) generates the potential solutions, such as in Algorithm 1, creating the minimum space to satisfy spatial arguments. Once a solution set is found, a uniform random sampling is applied to the set to select a single output. For N objects, the random sampling would occur N times on the generated solution set.

Algorithm 1 LSCENE Position Function

AO = A list of all world objects given a the symbol of that object, terrain, skybox, etc.
AC = A list of all spatial constraints to apply to the objects potential spawn locations.
while AO **do** ▷ Iterate over all objects defined under the given symbol
  $CB \leftarrow$ max sized 3D bounding box ▷ Start potential spawn location large
  while AC **do**
    $CB \leftarrow AC_N(CB)$ ▷ Shrink the 3D bounding box to fit the constraint
  end while
  AO_N.Position$\leftarrow$Random 3D Point in CB ▷ After all spatial constraints applied, generate random point
  if AO_N is Overlapping **then**
    Check AO_N constraints vs Overlapping Object
    Delete, hide, move, or allow overlapping objects ▷ Based on constraints of overlapping and current obj.
  end if
end while

Last, a simple parsing algorithm, see Algorithm 2, is defined and used to convert the JSON into C++ data types and function calls.

Algorithm 2 Parsing JSON into UE Data and C++ Functions

FL = A Map with string keys, and C++ U5 Function pointers as values (A reflection system to give C++ Functions a string reference name).
FC = A list of referenced Function pointers by the JSON
VC = A list of referenced variables by the JSON
LS = A List of the symbols in the language
JSON = Current LSCEN JSON object
CS = Current symbol context
Call Function: $f(JSON_{0,0}, null)$

```
while Node_{branch,0} ≠ null do                                   ▷ While more tree branches
    depth ← 0
    while Node_{branch,depth} ≠ null do                           ▷ While more tree nodes at branch N
        if CS ∈ LS then                                           ▷ Add Symbol Context to every function
            VC.Insert(CS)
        end if
        if Node_{branch,depth}.Key ∈ LS then
            f(Node_{branch,depth+1}.Value, Node_{branch,depth}.Key)   ▷ Call f Again with New Symbol Context
            branch ← branch + 1                                   ▷ Get other symbols on the same depth
        else if Node_{branch,depth}.Key ∈ FL then                 ▷ Add current key to Function list if it is Function List
            Func ← FL.At(Node_{branch,depth}.Key)
            FC.Insert(Func)
            depth ← depth + 1
        else
            VC.Insert(Node_{branch,depth}.Key)                    ▷ else, found a variable name or value
            depth ← depth + 1
        end if
    end while
    branch ← branch + 1
end while
```

REFERENCES

[1] "Gazebo." http://gazebosim.org/. (Accessed: 25 March 2023).
[2] "Robot Operating System (ROS)." https://ros.org. (Accessed: 25 March 2023).
[3] Roberts, M., Ramapuram, J., Ranjan, A., Kumar, A., Bautista, M. A., Paczan, N., Webb, R., and Susskind, J. M., "Hypersim: A photorealistic synthetic dataset for holistic indoor scene understanding," in [*ICCV*], (2021).
[4] Liang, J., Makoviychuk, V., Handa, A., Chentanez, N., Macklin, M., and Fox, D., "Gpu-accelerated robotic simulation for distributed reinforcement learning," (2018).
[5] "Unreal Engine." https://www.unrealengine.com/. (Accessed: 1 March 2023).
[6] "Unity." https://unity.com/. (Accessed: 25 March 2023).
[7] "COMSOL." https://www.comsol.com/. (Accessed: 1 March 2023).
[8] Oskooi, A. F., Roundy, D., Ibanescu, M., Bermel, P., Joannopoulos, J., and Johnson, S. G., "Meep: A flexible free-software package for electromagnetic simulations by the fdtd method," *Computer Physics Communications* **181**(3), 687–702 (2010).
[9] Rohde, M. M., Crawford, J., Toschlog, M. A., Iagnemma, K., Kewlani, G., Cummins, C. L., Jones, R. A., and Horner, D. A., "An interactive physics-based unmanned ground vehicle simulator leveraging open source gaming technology: progress in the development and application of the virtual autonomous navigation environment (vane) desktop," in [*Defense + Commercial Sensing*], (2009).

[10] Durst, P., Bethel, C., and Anderson, D., "A historical review of the development of verification and validation theories for simulation models," *International Journal of Modeling, Simulation, and Scientific Computing* **08** (01 2017).

[11] Nikolenko, S. I., "Synthetic data for deep learning," (2019).

[12] Lai, K.-T., Lin, C.-C., Kang, C.-Y., Liao, M.-E., and Chen, M.-S., "Vivid: Virtual environment for visual deep learning," in [*Proceedings of the 26th ACM International Conference on Multimedia*], *MM '18*, 1356–1359, Association for Computing Machinery, New York, NY, USA (2018).

[13] Kishore, A., Choe, T. E., Kwon, J., Park, M., Hao, P., and Mittel, A., "Synthetic data generation using imitation training," in [*Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV) Workshops*], 3078–3086 (October 2021).

[14] Wang, H., Wang, Q., Yang, F., Zhang, W., and Zuo, W., "Data augmentation for object detection via progressive and selective instance-switching," (2019).

[15] Georgakis, G., Mousavian, A., Berg, A. C., and Kosecka, J., "Synthesizing training data for object detection in indoor scenes," (2017).

[16] Haltakov, V., Unger, C., and Ilic, S., "Framework for generation of synthetic ground truth data for driver assistance applications," in [*Pattern Recognition*], Weickert, J., Hein, M., and Schiele, B., eds., 323–332, Springer Berlin Heidelberg, Berlin, Heidelberg (2013).

[17] Mayer, N., Ilg, E., Fischer, P., Hazirbas, C., Cremers, D., Dosovitskiy, A., and Brox, T., "What makes good synthetic training data for learning disparity and optical flow estimation?," *International Journal of Computer Vision* **126**, 942–960 (apr 2018).

[18] Saleh, F. S., Aliakbarian, M. S., Salzmann, M., Petersson, L., and Alvarez, J. M., "Effective use of synthetic data for urban scene semantic segmentation," (2018).

[19] Shrivastava, A., Pfister, T., Tuzel, O., Susskind, J., Wang, W., and Webb, R., "Learning from simulated and unsupervised images through adversarial training," (2017).

[20] McCormac, J., Handa, A., Leutenegger, S., and Davison, A. J., "Scenenet rgb-d: 5m photorealistic images of synthetic indoor trajectories with ground truth," (2017).

[21] Richter, S. R., Vineet, V., Roth, S., and Koltun, V., "Playing for data: Ground truth from computer games," (2016).

[22] Buck, A., Derek Anderson, J. F., Kerley, J., and Palaniappan, K., "Ignorance is bliss: flawed assumptions in simulated ground truth," in [*SPIE*], (2023).

[23] Gaidon, A., Wang, Q., Cabon, Y., and Vig, E., "Virtual worlds as proxy for multi-object tracking analysis," (2016).

[24] Marelli, D., Bianco, S., and Ciocca, G., "Ivl-synthsfm-v2: A synthetic dataset with exact ground truth for the evaluation of 3d reconstruction pipelines," *Data in brief* **29**, 105041 (April 2020).

[25] Degol, J., Lee, J. Y., Kataria, R., Yuan, D., Bretl, T., and Hoiem, D., "Feats: Synthetic feature tracks for structure from motion evaluation," in [*2018 International Conference on 3D Vision (3DV)*], 352–361 (2018).

[26] Tobin, J., Fong, R., Ray, A., Schneider, J., Zaremba, W., and Abbeel, P., "Domain randomization for transferring deep neural networks from simulation to the real world," (2017).

[27] Tremblay, J., Prakash, A., Acuna, D., Brophy, M., Jampani, V., Anil, C., To, T., Cameracci, E., Boochoon, S., and Birchfield, S., "Training deep networks with synthetic data: Bridging the reality gap by domain randomization," (2018).

[28] Raghavan, N., Chakravarty, P., and Shrivastava, S., "Sim2real for self-supervised monocular depth and segmentation," (2020).

[29] Behl, H. S., Baydin, A. G., Gal, R., Torr, P. H. S., and Vineet, V., "Autosimulate: (quickly) learning synthetic data generation," (2020).

[30] Ruiz, N., Schulter, S., and Chandraker, M., "Learning to simulate," (2019).

[31] Jiang, Y., Zhang, H., Zhang, J., Wang, Y., Lin, Z., Sunkavalli, K., Chen, S., Amirghodsi, S., Kong, S., and Wang, Z., "Ssh: A self-supervised framework for image harmonization," (2021).

[32] Cong, W., Niu, L., Zhang, J., Liang, J., and Zhang, L., "Bargainnet: Background-guided domain translation for image harmonization," (2021).

[33] Dwibedi, D., Misra, I., and Hebert, M., "Cut, paste and learn: Surprisingly easy synthesis for instance detection," (2017).
[34] Niu, L., Cong, W., Liu, L., Hong, Y., Zhang, B., Liang, J., and Zhang, L., "Making images real again: A comprehensive survey on deep image composition," (2022).
[35] Perlin, K., "An image synthesizer," in [*Proceedings of the 12th Annual Conference on Computer Graphics and Interactive Techniques*], *SIGGRAPH '85*, 287–296, Association for Computing Machinery, New York, NY, USA (1985).
[36] Church, E. M. and Semwal, S., "Simulating trees using fractals and l-systems," (2006).
[37] Prusinkiewicz, P., Hanan, J., Hammel, M., and Mech, R., "L-systems: from the theory to visual models of plants," (2001).
[38] Olsen, J., "Realtime procedural terrain generation," (2004).
[39] Kelly, G. and McCabe, H., "A survey of procedural techniques for city generation," **14** (01 2006).
[40] Booth, M., "The ai systems of left 4 dead," in [*Artificial Intelligence and Interactive Digital Entertainment Conference*], Stanford University (2009).
[41] Team, O. E. L., Stooke, A., Mahajan, A., Barros, C., Deck, C., Bauer, J., Sygnowski, J., Trebacz, M., Jaderberg, M., Mathieu, M., McAleese, N., Bradley-Schmieg, N., Wong, N., Porcel, N., Raileanu, R., Hughes-Fitt, S., Dalibard, V., and Czarnecki, W. M., "Open-ended learning leads to generally capable agents," (2021).
[42] Chebotar, Y., Handa, A., Makoviychuk, V., Macklin, M., Issac, J., Ratliff, N., and Fox, D., "Closing the sim-to-real loop: Adapting simulation randomization with real world experience," 8973–8979 (05 2019).
[43] Valtchev, S. and Wu, J., "Domain randomization for neural network classification," (09 2020).
[44] Chen, X., Hu, J., Jin, C., Li, L., and Wang, L., "Understanding domain randomization for sim-to-real transfer," (2022).
[45] Truong, J., Rudolph, M., Yokoyama, N. H., Chernova, S., Batra, D., and Rai, A., "Rethinking sim2real: Lower fidelity simulation leads to higher sim2real transfer in navigation," in [*6th Annual Conference on Robot Learning*], (2022).
[46] Alvey, B., Anderson, D. T., Keller, J. M., Buck, A., Scott, G., Ho, D., Yang, C., and Libbey, B., "Improving explosive hazard detection with simulated and augmented data for an unmanned aerial system," in [*SPIE*], (2021).
[47] 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).
[48] Buck, A., Deardorff, M., Murray, B., Anderson, D., Keller, J., Popescu, M., Ho, D., and Scott, G., "Estimating depth from a single infrared image," in [*MSS*], (2023).
[49] Akers, J., Buck, A., Camaioni, R., Anderson, D., Luke, R., Keller, J., Deardorff, M., and Alvey, B., "Simulated gold standard for quantitative evaluation of monocular vision algorithms," in [*SPIE*], (2023).
[50] Buck, A., Anderson, D., Camaioni, R., Akers, J., Luke, R., and Keller, J., "Capturing uncertainty in monocular depth estimation: Towards fuzzy voxel maps," in [*FUZZ-IEEE*], (2023).
[51] Redmon, J., Divvala, S., Girshick, R., and Farhadi, A., "You only look once: Unified, real-time object detection," (2015).
[52] Redmon, J. and Farhadi, A., "Yolo9000: Better, faster, stronger," (2016).
[53] Redmon, J. and Farhadi, A., "Yolov3: An incremental improvement," (2018).
[54] "Infinite Studio." https://infinitestudio.software/. (Accessed: 25 March 2023).
[55] "Quixel." https://quixel.com/. (Accessed: 25 March 2023).
[56] "AirSim." https://github.com/microsoft/AirSim. (Accessed: 25 March 2023).
[57] "UE MRQ Render Passes." https://docs.unrealengine.com/4.27/en-US/AnimatingObjects/Sequencer/HighQualityMediaExport/RenderSettings/RenderPasses/. (Accessed: 25 March 2023).