A OpenXLand Components

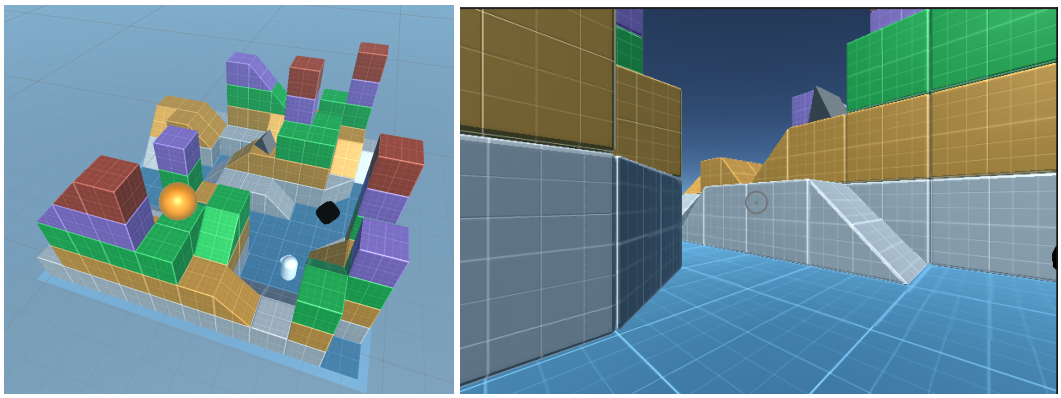


Figure 6: **Left:** An example OpenXLand environment. In this environment, the agent receives reward when the orange sphere makes contact with the blue pyramid. We see that the orange sphere is elevated, and therefore the agent must find it and use the ramps to access it. As to the blue pyramid, we do not see it because it is not there: the agent must first get the orange sphere near the black rounded cube first to spawn one. This environment also contains a grey pyramid that serves as a distraction. Importantly, if the agent brings the grey pyramid near the black rounded cube, both will disappear, making it impossible for the agent to spawn a blue pyramid and subsequently obtain its reward. **Right:** The agent's perspective at initialization. The agent does not see the orange sphere, and so will have to search for it. However, the agent does see the grey pyramid, which could act as a distraction. As pointed out in [21], solving tasks in OpenXLand requires many skill such as navigation, exploration, experimentation, and avoiding irreversible transitions. However, since the innumerable tasks are composed of a small number of components, the agent can learn to reuse previously acquired knowledge to improve performance.

OpenXLand is an open-source implementation of the environment frameworks known as XLand [22] and XLand 2.0 [21]. OpenXLand generates 3D worlds from building blocks known as *metatiles*. In the world, agents are expected to solve tasks defined by *production rules*, which trigger based on the state of the agent and objects in the environment. Figure 6 shows an example of such a world and task.

Below, we go over specific details of the metatiles and the production rules.

A.1 Metatiles

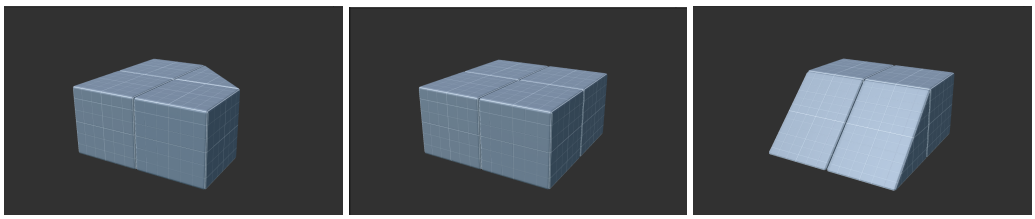


Figure 7: Three examples of metatiles. On the left, we see a platform with a rounded corner. The middle contains a square platform. The right contains a ramp, which the agent can use to traverse multi-level floor topologies.

The OpenXLand framework uses procedural generation to create diverse environments from simple building blocks we call *metatiles*. During generation, the environment is filled with metatiles. These metatiles are designed to create interesting multi-level floor topologies: agents can use ramps to access platforms (see Figure 7). Metatiles for OpenXLand can be found in the Resources/GameObjects/XLand/MetaTiles subfolder of the Unity asset folder. There are 13 total

Name	Description	Sampling Weight
RampInset	2 tile wide ramp with tiles on both sides	1500
Ramp1Edge	2 tile wide ramp with tiles on one side	750
RampNoEdge	2 tile wide ramp with no tiles the sides	750
Corner	4 tile platform with one corner rounded	1000
PlatformNoEdge	4 tile platform with no internal edges	100
Platform1Edge	4 tile platform with a single internal edge	100
Platform2Edge	4 tile platform with a two internal edges at the corner	100
Platform4Edge	4 tile platform with a four internal edges	100
AirPocket	4 empty tiles	100
SmallRamp	A minimal 1 tile wide ramp	450
Air	A empty tile	45
WhiteTile	A single tile	4
AirHelper	A tile that allows an empty tile to connect to internal edges	0.0004

Table 1: Metatiles used in OpenXLand. The sampling weight is used to calculate how likely a tile is to be sampled. U3 makes it possible to easily adjust these weights to enrich environment generation dynamics.

metatiles distributed across two pools: a medium pool with a weight of 10, and a small pool with a weight of 1. This means that the procedural generation algorithm will attempt to place medium metatiles much more frequently than small metatiles. A full list of metatiles and their relative weights can be found in Table 1 However, the small metatiles still serve an important role as they can fill in gaps where medium metatiles no longer fit. Note that AirHelper is meant to be used only when no other tiles are available.

A.1.1 The Face Matching Matrix

The logic for how metatiles fit together is defined by the face matching matrix. This adjacency matrix allows users to define which faces can be placed next to other faces, and with what weight. U3 will attempt to place metatiles at frequencies that respect both the face matching matrix frequencies and the metatile pool weights, however, due to the complexity of the generation process there is a fair amount of variance in the actual number of each metatile and face-face pairs placed.

OpenXLand uses 7 face types. Bedrock denotes the bottom of tiles, so Air and ramps cannot be placed below it. Similarly, Surface denotes the top of a tile and can match with any type of face, but attempts to avoid Air (0.05 weight). Air and Ground are generic face types of empty and solid faces respectively. InsideEdge attempts to connect to other InsideEdge faces, to create larger platforms. RampTop and RampBottom attempt to connect in a mutually exclusive manner to make sure that multiple ramps can chain together in the correct way.

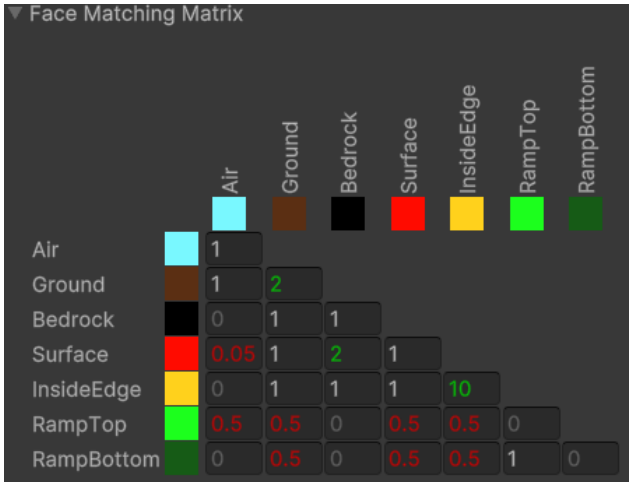
A.1.2 Wave Function Collapse Algorithm

U3 implements a 3D wave function collapse algorithm with a several optimizations. The original wave function collapse algorithm tracks all possible configurations of each world voxel to calculate the entropy of the voxels. We simplifi

the entropy of all world voxels simultaneously. In our modified version we only consider the number of metatile configurations (entropy)

A.2 Production Rules

A key element of XLand 2.0 is *production rules*, which define compositional tasks that agents are meant to complete. Agents 1) manipulate production rule objects 2) to satisfy production rule conditions, which 3) trigger production rule actions that transition the state. The goal of the agent is to trigger these production rules until the environment reaches a goal state and the agent receives reward.



Face Types
Air
Ground
Bedrock
Surface
InsideEdge
RampTop
RampBottom

Figure 8: **Left:** The face matching matrix between face types in OpenXLand. **Right:** The face types in the adjacency matrix. Each metatile is made of tiles, which have faces, and the adjacency matrix determines which faces can be neighbors. The adjacency matrix constrains the set of permissible metatiles that can be placed at a given location during the environment generation process.

In Table 2, we outline the production rule objects by shape and color, as well as the production conditions and the production rule actions. Naturally, each set can be extended.

Object Shapes	Object Colors	Conditions	Actions
Cube	red	NEAR	SPAWN
Sphere	green	CONTACT	SWAP
Pyramid	blue	HOLD	REMOVE
Cylinder	purple	PICKUP	REWARD
Cone	yellow	THROW	
Rounded Box	grey	DROP	
Rounded Rectangle	white		
	etc.		

Table 2: A table containing the main components of OpenXLand's production rule system. Our set of objects, which is specified by shape and color, is similar in nature to that of XLand 2.0 [21], though with 7 objects and over 20 colors. As for the conditions We extend those in Team et al. [21] with the new conditions "drop", "pickup" and "throw". Also, whereas in XLand 2.0, objects that are part of production rule conditions disappear when the condition is fulfilled, we include the possibility of objects remaining in the environment. The "spawn" action creates a new object, the "swap" action swaps the production rule condition's subject, object or both for a new object, the "remove" action simply removes an item from the environment, and the "reward" action gives reward to the agent. Therefore, our production rules go beyond forcibly removing the condition's objects or swapping them for new ones.

A.2.1 Production Rule Objects

Our production rule objects are quite similar to those of Team et al. [21]: as us, they use cubes, spheres and pyramids (not counting walls), but we also include other objects such as cylinders, cones, and rounded boxes and rectangles. Team et al. [21] uses three colors: black, purple and yellow. We see no reason a priori why there shouldn't be more colors, so we include over 20 colors, a few of

which can be found in 2. This raises the unique number of object-shape pairs from 9 to over 150. Naturally, we can restrict or extend the set of colors. Furthermore, although OpenXLand's objects are identified with simple colors and shapes, it is possible to use U3 to identify objects with different characteristics.

A.2.2 Production Rule Conditions

We implement the following conditions from Team et al. [21]: "near", "contact" and "hold". We plan on implementing the final condition, "see" in future work. On top of this, we implement three new conditions: "pickup", "throw" and "drop", which we implement as callback functions that trigger based on the agent's behaviour. The agent can only hold (and therefore pickup, throw and drop) one object at a time.

A.2.3 Production Rule Actions

In Team et al. [21], production rules and goal states are represented as separate concepts. We find it more amenable to represent them as instances of the same concept by turning goal states into production rules that reward the agent as an action. This allows us to represent production rules and goal states as part of the same language. We also include in our production rule actions the possibility of preserving the objects present in the production rule conditions, which can allow for distinct production rules that repeatedly depend on the same object.

B Implementing MemoryMaze: A Walkthrough

MemoryMaze [18] differs from OpenXLand in a few keys ways: First, the world is flat, so we only need a height of 1. Second, production rules are not procedurally generated, but are instead attached to "goal" objects spawned throughout the world. Third, only 1 "goal" is active at a time. Fourth, there is a UI element that displays the current "goal".

To illustrate the ability of U3 to render more naturalistic worlds, we use a low polygon forest pack from the Unity asset store. Thus, we modify the original MemoryMaze by replacing the walls with rocks, and the goals with forest mushrooms of various colors.

Our MemoryMaze starts by procedurally generating the flat world using the same metatile system as OpenXLand. We want goals to be spread out and only spawn in small clearings. Thus, we define a new class (ProductionRuleSpawn) that defines locations where goals can spawn, and add it to the Clearing metatile itself. By adding a script into the metatile we are able to use the existing procedural generation system to add structured randomness to this new spawning logic. We also include custom spawn logic in the script to spawn only up to 5 goals, and make sure that each goal is a minimum distance from the previously spawned goals.

Further, we add logic to ProductionRuleSpawn such that a new production rule is created for each spawned goal. These rules all use a new NEAR condition that triggers when the agent is near the production rule's subject (in this case, the goal object). Because of the modular design of U3, this new condition can now be used in any newly generated OpenXLand environments as well. We also create a new REWARD_TOGGLE action that is a compound action of a REWARD plus new logic that deactivates the current production rule, and activates a different production rule at random. To implement this logic we also add a new flag to production rules that allows them to be turned on and off at run-time. In this way, as new environments are added to the U3 ecosystem, we expect that new and more powerful features will also be added to the base OpenXLand environments.

The final missing piece of the MemoryMaze environment is the UI element that shows the current goal. We adopt the same UI as the original with a colored border around the image. Unity is a powerful 3D game engine with native support for 3D rendering, UI, sprites, materials, etc, so creating this UI took only 1 function call into Unity's API (GUI.DrawTexture).

Figure 9: MemoryMaze environment observation. Our version of MemoryMaze uses naturalistic assets. We hope that harnessing the variety of assets available will help to increase agents' overall adaptability.

C Dataset Details

In this section, we go over the details of the datasets. We split the datasets into 6 world datasets 3, and 6 production rule datasets 4. We design the datasets with overlapping parameters to facilitate smooth curriculum learning. We filter the world datasets to include only those worlds that have a largest accessible area of over 50% of the total number of tiles in the X-Z plane.

We find the largest accessible area by first constructing a local, directed connectivity graph using the specific configuration of each voxel. For simplicity, we consider only the cardinal connections between voxels, and only connections in the width-length plane (we ignore connections within a single column of voxels). Two voxels are connected bi-directionally if they are at the same terrain height and higher voxels are connected unidirectionally to lower tiles - representing the agent's ability to fall to the lower level. Ramps serve as the sole method for connections from a lower layer to the next highest layer. Once the connectivity graph is constructed, the largest accessible area is found by computing the largest strongly connected sub-component.

Distractors' positions in the production rule chains heavily influence their impact. For example, if the agent has already completed the step that requires a Yellow Cube, then removing the Yellow Cube is merely a distraction. However, if the Yellow Cube is still required, then its removal creates a dead end. Due to this complexity, we do not directly consider distractors vs dead ends in our difficulty measure, but note that as more distractors are added the probability of dead ends also increases.

D Training Results

To demonstrate the learnability of our environment, we randomly select a task from the easy dataset and applied the Soft Actor-Critic (SAC) [11] method. Image observations are processed using a convolutional neural network for both the actor and Q networks, each comprising approximately 2 million parameters. The SAC agent is trained for 75 million environment steps, utilizing a replay buffer that stored 1 million transitions. While these parameter choices may appear modest compared to those in the Ada paper [21] and may not fully exploit emergent meta-learning capabilities, our primary aim is to underscore the learnability of our environment, not to recreate their agent.

Name	Width and Length		Height	
	$\bar{x}$	σ	$\bar{x}$	σ
easy_low	5	2	1	1
easy_high	5	2	3	2
medium_low	10	3	1	1
medium_high	10	3	3	2
hard_low	15	5	1	1
hard_high	15	5	3	2

Table 3: World Datasets. These values were used to define Gaussian distributions used to sample parameters during dataset construction. We note that the harder we consider environments to be, the bigger they are.

Name	Chain Length and Initial Objects		Distractors	
	$\bar{x}$	σ	$\bar{x}$	σ
short_few	1	1	0	1
short_many	1	1	1	0.5
middle_few	3	1.5	0	1
middle_many	3	1.5	2	1.5
long_few	5	3	0	1
long_many	5	3	3	2.5

Table 4: Production Rule Datasets. These values were used to define the Gaussian distribution used to sample parameters during dataset construction. Since we sample the production rule chain lengths from distributions, the datasets of shorter chains will still contain long chains, although the dataset will be dominated by shorter, less complicated tasks. As the average length of chains increases throughout the datasets, the number of distractors also increases.

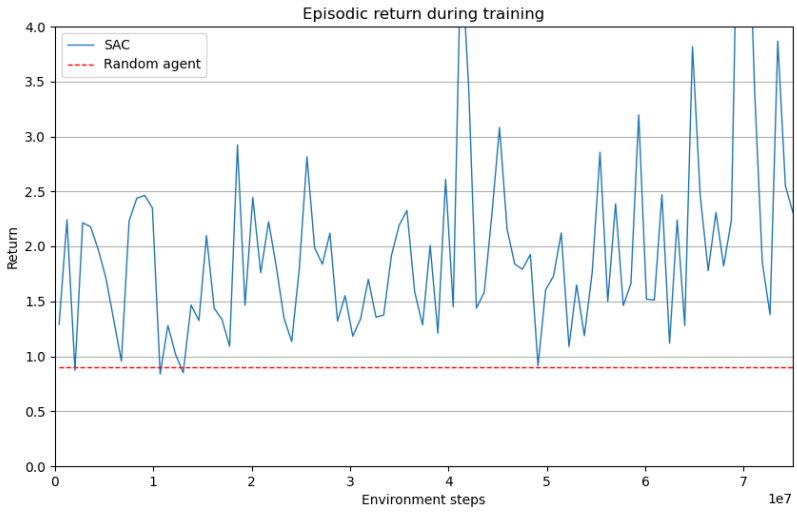


Figure 10: Episodic return during single environment training of the SAC agent compared to a random agent. We train the SAC agent for 75 million environment steps. This required approximately 48 hours on a single A100 GPU, a small fraction of the 53760 TPUv3 device hours it took to train Ada [21]. The episodic return for the SAC agent shows significant fluctuations but generally demonstrates a higher return compared to the random agent baseline, represented by the red dashed line.

Figure 10 illustrates that the episodic return of our learning agent increases throughout training, surpassing a random agent baseline's performance. We also extend our evaluation to the entire easy dataset, conducting 8 trials per episode on different tasks. Interestingly, our learning agent exhibits minimal improvement over random agents in this broader context, indicating the limitations of conventional methods. We observe similar outcomes when applying the method to medium datasets, where random agents fail to accrue any rewards, underscoring the challenge of reward collection within these datasets.

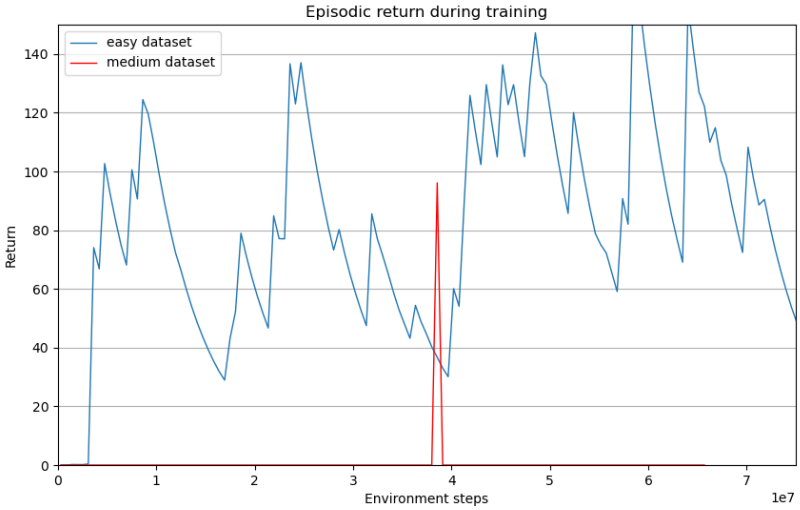


Figure 11: Episodic return during training using SAC on the whole dataset with 8 trials per episode and changing the task each episode. Collecting reward on the medium dataset is hard showing the limitations of the conventional methods. Collecting reward on easy dataset is easier, but meta-learning requires more advanced methodologies.

E Resources

Most of the resources for this project were spent on implementing the baselines. The CPU compute spent to generate the envs amounts to 200 hours. The GPU compute spent to train the baselines amounts to around 140 hours of NVIDIA A100-SXM4-80GB.

F URL to Dataset and Metadata

Datasets: `https://huggingface.co/datasets/cerc-aai/u3_datasets`

U3 source code: `https://github.com/CERC-AAI/u3/`

Croissant Metadata: `https://huggingface.co/api/datasets/cerc-aai/u3_datasets/croissant`

We plan to fully open-source the code for U3, including the Unity environment framework, the python training code and the pipeline. Using U3 still requires installing Unity, which impinges on reproducibility. As stated in Section 7, we plan on containerizing the environment to democratize access to U3.

G Datasheet

G.1 Motivation

1. **For what purpose was the dataset created?** *U3 and its associated OpenXLand datasets were created to promote the open development of foundation models for RL and meta-RL.*
2. **Who created the dataset and on behalf of which entity?** *U3 and its datasets were created by the researchers listed in the author list.*
3. **Who funded the creation of the dataset?** *The main funding bodies include the Canada Excellence Research Chairs Program, as well as the resources of the INCITE program award "Scalable Foundation Models for Transferable Generalist AI" provided by Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725*

G.2 Distribution

1. **Will the dataset be distributed to third parties outside of the entity (e.g., company, institution, organization) on behalf of which the dataset was created?** *The U3 framework and its datasets will be publicly available.*
2. **How will the dataset be distributed (e.g., tarball on website, API, GitHub)?** *The datasets will be distributed through Hugging Face, while the code will be made available on Github.*
3. **Have any third parties imposed IP-based or other restrictions on the data associated with the instances?** *No.*
4. **Do any export controls or other regulatory restrictions apply to the dataset or to individual instances?** *No.*

G.3 Maintenance

1. **Who will be supporting/hosting/maintaining the dataset?** *The CERC-AAI Lab at Université de Montréal will maintain the U3 framework and its datasets.*
2. **How can the owner/curator/manager of the dataset be contacted (e.g., email address)?** *The managers of the dataset can be contacted at connor.brennan@mila.quebec, andrew.williams@mila.quebec and irina.rish@gmail.com*
3. **Is there an erratum?** *No. We will release one in the future if necessary.*
4. **Will the dataset be updated (e.g., to correct labeling errors, add new instances, delete instances)?** *Yes, the datasets will be updated if necessary.*
5. **If the dataset relates to people, are there applicable limits on the retention of the data associated with the instances (e.g., were the individuals in question told that their data would be retained for a fixed period of time and then deleted?)** N/A
6. **Will older versions of the dataset continue to be supported/hosted/maintained?** N/A
7. **If others want to extend/augment/build on/contribute to the dataset, is there a mechanism for them to do so?** *U3 welcomes contributors on GitHub. Should the need arise to extend the datasets on Hugging Face, we will establish a procedure for doing so.*

G.4 Composition

1. What do the instances that comprise the dataset represent **(e.g., documents, photos, people, countries?)** *The instances that comprise the datasets are JSON datafiles that describe the necessary information to instantiate RL environments in Unity.*
2. **How many instances are there in total (of each type, if appropriate)?** *Six files of one million environment instances, six files of one million production rule instances, for a total of twelve million instances spread across 12 files.*

3. **Does the dataset contain all possible instances or is it a sample of instances from a larger set?** *Is it a subset of the effectively infinite number of instances that U3 can generate.*
4. **Is there a label or target associated with each instance?** *Each instance contains the label information necessary to instantiate a Unity environment.*
5. **Is any information missing from individual instances?** *No.*
6. **Are there recommended data splits (e.g., training, development/validation, testing)?** *No.*
7. **Are there any errors, sources of noise, or redundancies in the dataset?** *The instances are sampled with replacement from a procedural generation process, so yes there can be redundancies.*
8. **Is the dataset self-contained, or does it link to or otherwise rely on external resources (e.g., websites, tweets, other datasets)?** *The datasets rely on access to a Unity instance*
9. **Does the dataset contain data that might be considered confidential?** *No.*
10. **Does the dataset contain data that, if viewed directly, might be offensive, insulting, threatening, or might otherwise cause anxiety?** *No.*

G.5 Collection Process

1. **How was the data associated with each instance acquired?** *We procedurally generated environments in Unity and collected the necessary information to recreate them using the U3 framework.*
2. **What mechanisms or procedures were used to collect the data (e.g., hardware apparatus or sensor, manual human curation, software program, software API)?** *We used the Unity Engine and Python, as well as GPUs for training RL baselines.*
3. **Who was involved in the data collection process (e.g., students, crowdworkers, contractors) and how were they compensated (e.g., how much were crowdworkers paid)?** *Students, paid through the funding sources described above.*
4. **Does the dataset relate to people?** *No.*
5. **Did you collect the data from the individuals in question directly, or obtain it via third parties or other sources (e.g., websites)?** N/A

G.6 Uses

1. **Has the dataset been used for any tasks already?** *No, the data has not been used, other than for running baselines.*
2. **What (other) tasks could the dataset be used for?** *The datasets could be used for RL-related tasks as standalone or as part of a blended dataset*
3. **Is there anything about the composition of the dataset or the way it was collected and preprocessed/cleaned/labeled that might impact future uses?** *The use of U3 and its datasets relies on access to the Unity Engine.*
4. **Are there tasks for which the dataset should not be used?** *No.*

H Author statement

We bear all responsibility in case of violation of rights.

We use the MIT License for our data.