

# 000 001 002 003 004 005 006 007 008 009 010 011 012 013 014 015 016 017 018 019 020 021 022 023 024 025 026 027 028 029 030 031 032 033 034 035 036 037 038 039 040 041 042 043 044 045 046 047 048 049 050 051 052 053 OMNI-EPIC: OPEN-ENDEDNESS VIA MODELS OF HUMAN NOTIONS OF INTERESTINGNESS WITH ENVIRONMENTS PROGRAMMED IN CODE

**Anonymous authors**

Paper under double-blind review

## ABSTRACT

Open-ended and AI-generating algorithms aim to continuously *generate* and *solve* increasingly complex tasks indefinitely, offering a promising path toward more general intelligence. To accomplish this grand vision, learning must occur within a vast array of potential tasks. Existing approaches to automatically generating environments are constrained within manually predefined, often narrow distributions of environments, limiting their ability to create *any* learning environment. To address this limitation, we introduce a novel framework, OMNI-EPIC, that augments previous work in Open-endedness via Models of human Notions of Interestingness (OMNI) with Environments Programmed in Code (EPIC). OMNI-EPIC leverages foundation models to autonomously generate code specifying the next learnable (i.e., not too easy or difficult for the agent’s current skill set) and interesting (e.g., worthwhile and novel) tasks. OMNI-EPIC generates both environments (e.g., an obstacle course) and reward functions (e.g., progress through the obstacle course quickly without touching red objects), enabling it, in principle, to create any simulatable learning task. We showcase the explosive creativity of OMNI-EPIC, which continuously innovates to suggest new, interesting learning challenges. We also highlight how OMNI-EPIC can adapt to reinforcement learning agents’ learning progress, generating tasks that are of suitable difficulty. Overall, OMNI-EPIC has the potential to endlessly create learnable and interesting environments, further propelling the development of self-improving AI systems and AI-Generating Algorithms. Project website with videos: <https://dub.sh/omni-epic>.

## 1 INTRODUCTION

In recent years, the field of artificial intelligence (AI) has witnessed groundbreaking achievements, particularly with advancements in reinforcement learning (RL) (Akkaya et al., 2019; Silver et al., 2016; Colas et al., 2019) and foundation models (FMs) (Brown, 2020; Radford et al., 2019). Yet, the most ambitious goal of AI — building generalist agents capable of autonomously perceiving, reasoning, deciding, and acting within complex environments — remains a formidable challenge. The creation of general-purpose agents promises immense societal benefits, provided we can address the critical safety and existential risks involved (Bengio et al., 2023), including those unique to open-ended and AI-generating algorithms (AI-GAs) (Clune, 2019; Ecoffet et al., 2020). While researchers have made substantial progress in developing powerful learning architectures (Hafner et al., 2023; Jaegle et al., 2021; Vaswani, 2017), these algorithms have been applied to a relatively narrow range of tasks due to the limited availability of diverse datasets. Compounding this issue, current models require extensive amounts of data and fine-tuning to be trained. Consequently, over the past decade, the bottleneck in advancing AI has shifted from improving learning algorithms to acquiring the necessary data to train them (Jiang et al., 2023). In other words, the main challenge has become: how to effectively acquire large amounts of data for diverse tasks.

A fundamentally different approach involves developing open-ended algorithms that continuously generate and solve new challenges endlessly (Stanley et al., 2017). The objective of open-ended algorithms is to ignite an explosion of creativity and complexity in a computer, akin to the processes observed in biological evolution and human culture, including science and technology. Replicating open-ended evolution *in silico* extends beyond understanding biological life or developing engaging

054 simulations. It delves into fundamental questions about the nature of creativity, the potential for  
 055 machines to exhibit life-like characteristics, and the emergence of general intelligence. Given that  
 056 biological evolution is the only known process that has successfully created general intelligence, its  
 057 principles hold invaluable insights for advancing the field of AI, and creating AI-GAs (Clune, 2019).

058 For open-ended algorithms and AI-GAs to succeed, they must operate within a *vast* task space capable  
 059 of generating an infinite array of potential challenges. Previous open-ended algorithms (Sudhakaran  
 060 et al., 2024; Wang et al., 2023a; 2019; 2020; Zhang et al., 2023) have been applied to limited domains  
 061 with specific types of worlds (e.g., obstacle courses) and confined to predefined parameterizations  
 062 (e.g., obstacle size and type), restricting their potential to exhibit true open-endedness. Achieving  
 063 *Darwin Completeness* — the potential to generate *any possible learning environment* — is essential  
 064 for realizing the full potential of AI-GAs (Clune, 2019). However, Darwin Completeness presents a  
 065 significant challenge: the ability to generate infinitely many environments makes the search potentially  
 066 intractable, or even impossible. Exploring this immense environment search space may involve  
 067 endlessly generating trivial, redundant, or overly complex tasks that do not effectively contribute to  
 068 the agent’s learning progress. The main challenges are to ensure that generated environments are  
 069 novel, interesting, solvable, and appropriately matched to the current capabilities of the learning  
 070 agents. Biological evolution required an unfathomable amount of computation over billions of years  
 071 to produce intelligence. Therefore, one key scientific challenge is to determine how we can optimize  
 072 the process of generating and solving interesting tasks within a Darwin Complete environment search  
 073 space, so as to create an AI-GA given the computational capabilities we expect to have in the future.

074 Previous work in Open-endedness via Models of human Notions of Interestingness (OMNI) (Zhang  
 075 et al., 2023) leverages FMs to improve open-ended learning by focusing on tasks that are both  
 076 learnable and interesting. However, OMNI, like all prior open-ended works (Sudhakaran et al.,  
 077 2024; Wang et al., 2019; 2020; Zhang et al., 2023; Wang et al., 2023a), was confined to generating  
 078 tasks within a narrow environment search space, inhibiting the generation of *any* possible learning  
 079 environment. This paper introduces a novel framework, OMNI-EPIC, that augments OMNI with  
 080 Environments Programmed in Code (EPIC). OMNI-EPIC utilizes FMs to choose the next interesting  
 081 and learnable task and subsequently generate environment code to enable the agent to learn how  
 082 to solve that task. Our approach generates not only the simulated world but also the reward and  
 083 termination functions, allowing it, in principle, to create any simulatable task. We take advantage of  
 084 pre-existing simulators and OMNI-EPIC writes code to create tasks within it. For example, if the  
 085 task is to kick a ball to hit a moving target, OMNI-EPIC would generate the environment code to  
 086 simulate the physics, the agent, the ball, and the moving target, rewarding the agent when the ball  
 087 hits the target. Conversely, for a task involving maneuvering the ball around a moving target, the  
 088 simulated world remains the same, but the reward function could differ, penalizing the agent for any  
 089 contact with the target. A model of interestingness (MoI) is employed both when generating the next  
 090 task and checking if any newly proposed task is interestingly new compared to similar ones in the  
 091 archive. Finally, we introduce a success detector that can automatically determine whether the agent  
 092 has successfully completed any proposed task.

093 Our vision is for this algorithm to generate any code, including installing and modifying any existing  
 094 simulator, or even writing the code for a new simulator. Given that the programming language used  
 095 here (Python) is Turing complete, OMNI-EPIC could potentially create any computable environment  
 096 (e.g., logic and math problems to quests in virtual worlds, such as building a computer in Minecraft).  
 097 As a first step toward this ambitious goal, in this work, we constrain our method to write code for  
 098 one simulator, namely PyBullet (Coumans & Bai, 2016). By continuously generating learnable and  
 099 interesting environments, OMNI-EPIC advances the development of self-improving AI systems,  
 bringing us closer to achieving Darwin Completeness and realizing AI-GAs.

## 100 2 RELATED WORK

101 **Unsupervised Environment Design.** Unsupervised environment design has garnered increasing  
 102 interest in RL. Several works have explored the development of auto-curricula to perpetually generate  
 103 new training environments for agents (Dennis et al., 2020; Jiang et al., 2021; Parker-Holder et al.,  
 104 2022; Samvelyan et al., 2023; Wang et al., 2019; 2020). However, a significant limitation of these  
 105 methods is their reliance on predefined or manually curated distributions of tasks or environment  
 106 parameters (Dennis et al., 2020; Heess et al., 2017; Jiang et al., 2021; Parker-Holder et al., 2022;  
 107 Samvelyan et al., 2023; Wang et al., 2019; 2020), inhibiting the generation of any possible learning

108 environment. Regret-based approaches (Jiang et al., 2021; Parker-Holder et al., 2022; Samvelyan  
 109 et al., 2023) prioritize tasks with high regret, measured by the difference between the highest known  
 110 return and the mean return across simulations. Alternative methods calculate learning progress by  
 111 measuring the difference in the agent’s task success rates across training steps (Kanitscheider et al.,  
 112 2021). By focusing on tasks with high learning progress, these approaches aim to guide the agent’s  
 113 learning towards the most promising areas of the task space (Oudeyer et al., 2007; Oudeyer & Kaplan,  
 114 2007; Baranes & Oudeyer, 2013). However, a critical challenge remains in distinguishing which  
 115 environments are genuinely interesting (Jiang et al., 2023; Zhang et al., 2023; Colas et al., 2022). In  
 116 the vast space of any task describable in natural language, there may be countless learnable but not  
 117 meaningful environments (e.g., kicking a ball into a goal at slightly different positions). Inspired  
 118 by Zhang et al. (2023), OMNI-EPIC uses human notions of interestingness distilled into FMs to  
 119 generate environments that are not only learnable, but also interesting.

120 **Foundation Models for Environment Design.** Recent advancements in FMs have showcased  
 121 their remarkable ability to capture extensive knowledge across diverse subjects by training on vast  
 122 text corpora (Bommasani et al., 2021). Consequently, this has sparked interest in applying FMs to  
 123 environment design. Ma et al. (2023) utilize FMs to generate code for reward functions while Wang  
 124 et al. (2023b) employ FMs to generate simulation environments and expert demonstrations. However,  
 125 these methods do not build upon the agent’s previous performance on different tasks, and hence lack  
 126 an auto-curriculum that can provide an endless stream of environments and tasks. In procedural  
 127 content generation (Shaker et al., 2016; Juliani et al., 2019; Justesen et al., 2018), Sudhakaran et al.  
 128 (2024) and Todd et al. (2023) fine-tune FMs to create domain-specific levels. Bruce et al. (2024)  
 129 propose training a world model to generate game environments. These approaches focus on level  
 130 creation but do not train agents or adapt the difficulty based on the agent’s performance or learning  
 131 progress. Zala et al. (2024) generate environments as a curriculum to learn a fixed set of tasks but  
 132 is limited in its ability to generate truly open-ended environments and adapt to the agent’s evolving  
 133 capabilities. Wang et al. (2023c) rely on a predefined set of objects and is limited in reflecting  
 134 real-world dynamics in its simulation. Despite significant progress in applying FMs to environment  
 135 design, opportunities remain for further exploration, such as integrating the generative capabilities  
 136 of FMs with adaptive auto-curricula. OMNI-EPIC addresses this challenge, aiming to unlock the  
 potential for truly open-ended and effective learning environments.

137 **Foundation Models in Open-Endedness.** The field of open-endedness seeks to create algorithmic  
 138 systems that produce never-ending innovation (Stanley et al., 2017). There has been increasing  
 139 interest in leveraging FMs to generate intelligent variations for code or text in evolutionary algorithms.  
 140 However, these approaches are often confined to a fixed archive with hand-crafted characteris-  
 141 tics (Bradley et al., 2023; Ding et al., 2023; Lehman et al., 2023; Lim et al., 2024). The core objective  
 142 of open-endedness algorithms is to generate and solve an endless stream of tasks. One way to  
 143 do so is to keep an ever-expanding archive of tasks or solutions. Zhang et al. (2023) and Wang  
 144 et al. (2023a) have adopted FMs as a mechanism for auto-curricula, proposing both learnable and  
 145 interesting tasks for agent training. By restricting the agent’s interactions to a predefined range of  
 146 environmental conditions, these methods may hinder the development of truly adaptable and versatile  
 147 agents capable of handling the complexities of real-world scenarios. OMNI-EPIC addresses this  
 148 limitation by leveraging FMs to generate not only tasks but also the simulated worlds and reward  
 149 functions, potentially exposing agents to a wider range of challenges and learning opportunities.

### 150 3 METHOD

151 OMNI-EPIC leverages FMs, including large language models (LLMs) and vision-language models  
 152 (VLMs), to autonomously create learnable and interesting tasks for open-ended learning (Figure 1).  
 153 OMNI-EPIC maintains a growing task archive (Section 3.1) that catalogs successfully learned and  
 154 completed tasks, as well as unsuccessfully attempted ones. The task generator (Section 3.2) uses  
 155 information from the archive about what has been learned and what has not, proposing the next  
 156 *interestingly new* task, described in natural language, for the agent to attempt. Because the model has  
 157 distilled a sense of what is interesting from training on internet data, it has a MoI that emulates the  
 158 human capacity for making nuanced judgments of interestingness in open-ended learning (Zhang  
 159 et al., 2023). The task generator utilizes this MoI when generating tasks. These tasks are then  
 160 translated into environment code by an environment generator (Section 3.3), specifying the simulated  
 161 world and functions required for RL. The newly generated task and its environment code are assessed  
 by a second, post-generation MoI (Section 3.4), to ensure the task is indeed interesting given what

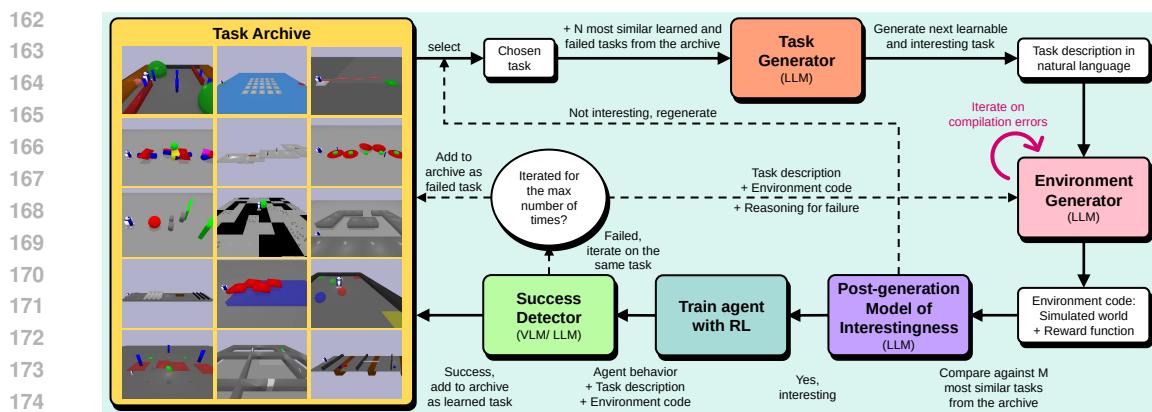


Figure 1: **OMNI-EPIC overview.** OMNI-EPIC continuously generates and solves new, interesting tasks in simulation. Our approach maintains a task archive of learned and failed tasks.

has come before (it compares the new task to the most similar tasks already in the archive with retrieval-augmented generation (Lewis et al., 2020). Tasks deemed interesting are then used to train an RL agent (Section 3.5). If deemed uninteresting, the task is discarded, and a new task is generated. After training, a success detector (Section 3.6) assesses whether the agent has successfully completed the task. Successfully completed tasks are added to the archive. Failed tasks are iterated upon a maximum number of times and added to the archive as failed tasks if the RL agents are not able to solve them. Then, the cycle of generating the next task restarts. Each component of OMNI-EPIC is explained in more detail below, and hyperparameters are shown in Appendix K. OMNI-EPIC’s iterative process ensures continuous generation and learning of new interesting tasks, forming a potentially never-ending growing collection of environments and learned agents.

### 3.1 TASK ARCHIVE

OMNI-EPIC maintains a continuously expanding archive of tasks, including successfully learned ones and those attempted but failed. Successful tasks serve as stepping stones for creating more complex yet learnable ones, while failed tasks provide insights into generating new tasks within the agent’s current capabilities. OMNI-EPIC uses past experiences to generate novel and diverse challenges, continuously pushing the boundaries of what’s already learned. The task archive is initialized with a few task description seeds in natural language (Appendix N). This archive is unbounded and can grow indefinitely as new tasks are generated and learned. Given the generality, interpretability, and universality of natural language and programs (Hopcroft, 2001), each task is represented by its natural language description and the corresponding environment, represented as executable code.

### 3.2 TASK GENERATOR

Open-ended algorithms require focusing on tasks that are both learnable (i.e., not too difficult or too easy for the agent to learn) and interesting (i.e., worthwhile and sufficiently novel). Previous attempts have resulted in pathologies when optimizing against definitions and quantifications of interestingness (Zhang et al., 2023). Inspired by Zhang et al. (2023), we harness FMs to model ineffable human notions of interestingness, gleaned from large text corpora of human-generated data. Here, the task generator is an LLM, which proposes novel task descriptions in natural language that are distinct from those already discovered while remaining learnable (full prompt in Appendix L.1).

To ensure the task generator remains open-ended and continuously suggests new and diverse tasks, it uses the content of the task archive as context. Given the limited context length of current LLMs, we retrieve a predefined number of tasks that are most similar to a randomly selected task from the archive (Appendix J). These retrieved tasks include both those that were successfully completed and those that were attempted but failed. These similar tasks serve as examples and are input into the LLM, which then generates the next learnable and interesting task. We opt for the most similar tasks rather than the most different ones to use previous tasks as stepping stones. As LLMs and FMs improve, we expect that a larger portion of the task archive, or potentially the entire archive, could be used as context. However, partial knowledge of stepping stones might be advantageous for creativity and diversity, much as human scientists and artists benefit from not being aware of

216 everything that has come before. The task generator outputs a natural language description of the  
 217 next task, crafted to be both achievable and interesting for the agent. This description serves as the  
 218 basis for the subsequent environment generation step, where the natural language task descriptions  
 219 are translated into executable code to create learning environments.  
 220

### 221 3.3 ENVIRONMENT GENERATOR

222 The environment generator, powered by an LLM, translates a given natural language task description  
 223 into executable (here, Python) code defining the learning environment. Appendix L.2 shows the  
 224 full prompt. This code includes specifications for creating the simulated world and functions  
 225 needed for RL (Sutton, 2018) based on the standard API Gymnasium (Towers et al., 2024): `reset`,  
 226 `step`, `reward`, and `terminated`. The `reset` function resets the environment to an initial state,  
 227 including setting up the initial positions and orientations of the agent and objects. The `step` function  
 228 updates the environment according to the simulated physics, the agent’s action, and any other dynamic  
 229 behaviors (e.g., moving platforms or activated doors). The `reward` function returns a scalar number,  
 230 whose cumulative maximization defines the task. The `terminated` function indicates whether the  
 231 agent has reached a terminal state.

232 For example, if the task is to “cross a bridge with moving segments”, a bridge should be created in the  
 233 simulated world. The `reset` function should initialize the agent at the start of the bridge and each  
 234 segment’s position. The `step` function should update the environment based on the agent’s actions  
 235 and each segment’s movement. The `reward` function should reward the agent’s progress across the  
 236 bridge, and the `termination` function should indicate when the agent falls off the bridge.

237 If compilation errors occur when generating the environment code, the errors (with the traceback) are  
 238 fed back into the environment generator, which then modifies and improves the environment code.  
 239 Appendix L.3 shows the full prompt. This loop is limited to a maximum of five iterations per task. If  
 240 the code still fails to compile after these attempts, the uncompiled code is discarded, and the task  
 241 generator proposes a new task, potentially one that is less complex or differently structured.

### 242 3.4 POST-GENERATION MODEL OF INTERESTINGNESS

244 While ideally all tasks proposed by the task generator should be interesting (owing to its MoI), the  
 245 limited context length of current FMs prevents the task generator from considering the entire task  
 246 archive at once. Furthermore, it is often easier for an FM to evaluate whether a solution is good  
 247 rather than generate a good solution (Bradley et al., 2023). OMNI-EPIC draws inspiration from the  
 248 dynamics of human culture. For example, researchers often study a specific subset of previous works  
 249 to inspire new ideas. Once a new idea is produced, it is crucial to check the literature to determine  
 250 whether the contribution is truly novel. If the idea is deemed interestingly new, it is published,  
 251 adding to the ever-growing archive of human knowledge. This process creates a growing set of  
 252 stepping stones to leap off of, which is a key ingredient of open-endedness (Stanley & Lehman, 2015).  
 253 OMNI-EPIC captures these important dynamics of open-ended algorithms, considering a small batch  
 254 of related stepping stones when creating a new task (Section 3.2) and then verifying its novelty  
 255 against the task archive. Given a newly generated task and its corresponding environment code, we  
 256 compare it against a predefined number of the most similar tasks from the archive (Appendix J).  
 257 The post-generation MoI then evaluates if the new task is interesting (e.g., novel, surprising, diverse,  
 258 worthwhile) (full prompt in Appendix L.4). If the task is deemed interesting, we proceed to train an  
 259 RL agent on it. If not, the task is discarded, and a new one is generated.

### 260 3.5 TRAINING AGENTS WITH REINFORCEMENT LEARNING

261 A key objective of open-ended learning is to enable agents to master an ever-expanding set of tasks.  
 262 To achieve this, OMNI-EPIC generates a diverse array of tasks along with their corresponding  
 263 environments programmed in code. Agents are then trained in these environments using RL to solve  
 264 the generated tasks. In this work, we utilize the PyBullet physics simulator (Coumans & Bai, 2016).  
 265 While any RL algorithm could be used, we employ DreamerV3 (Hafner et al., 2023). Appendix K  
 266 details the hyperparameters and compute resources used. Agents receive proprioceptive (joints  
 267 positions and velocities) and visual information (images of size  $64 \times 64 \times 3$ ). While OMNI-EPIC  
 268 can be applied to any robot (Appendix O), due to computational limitations, we demonstrate results  
 269 using an R2D2 robot with a discrete action space of six actions: do nothing, go forward, go backward,  
 rotate clockwise, rotate counterclockwise, and jump. R2D2’s simple action space allows the agent to  
 learn tasks more efficiently, requiring fewer time steps than agents with complex action spaces.

Following Wang et al. (2020), we train one specialist RL agent for each task. For tasks used to initialize the archive, the RL agents are trained from scratch. For newly generated tasks, the agent continue training from an existing policy previously trained on tasks in the archive. The trained policy is selected from a successfully completed task with the closest embedding to the new task (Appendix J). This approach allows the agents to build upon their existing knowledge and adapt more efficiently to challenges presented by the new tasks. Additionally, color variation in the environments is a form of domain randomization, as an RL agent trained on an environment with specific colors may not perform well in an identical environment with different colors (Tobin et al.).

### 3.6 SUCCESS DETECTOR

In this infinite task space of any task describable in natural language, an essential ingredient to training agents to learn any generated task is a universal reward function, which can evaluate if *any* proposed task has been completed or not. We employ a success detector, instantiated as an LLM or VLM, that assesses whether the agent has successfully completed the given task. Since our preliminary testing found that current VLMs are not yet accurate enough to be used as success detectors (Appendix C), we use code generated by LLMs for this purpose instead.

When using an LLM as the success detector, we ask the environment generator to generate an additional success-checking function `get_success`, alongside the environment code, that checks if the agent has successfully completed the task. The success-checking function serves a different purpose from the reward function. The reward function is designed to enable the RL agent to learn efficiently, which often results in a more complex function than the success-checking function. The complexity of the reward function arises from the need to shape rewards to facilitate optimal learning (Krakovna et al., 2020). Meanwhile, the success-checking function aims to evaluate whether the agent has accomplished the task. This function is less susceptible to reward hacking, as it does not affect how the agent learns. For example, consider the task of “run forward”. The success-checking function would evaluate whether the agent’s x-axis position has exceeded a certain threshold. In contrast, the reward function should encourage efficient learning by rewarding the agent’s x-axis velocity and promoting cyclic joint movements that resemble a natural running motion.

If a task is deemed to have been successfully learned by the trained agent, it is added to the archive. If not, the task is returned to the environment generator for modifications to aid in the agent’s learning (e.g., changing the reward function or making the physical world less complex). This feedback loop is repeated for a max number of attempts. If the task remains unlearned after these attempts, it is added to the archive as a failed task and a new task is generated.

## 4 LONG RUN WITH SIMULATED LEARNING

To illustrate the creative explosion of generated tasks, we run OMNI-EPIC without training RL agents, assuming all generated tasks can be successfully completed. This allows us to showcase a larger number of generated tasks, as training RL agents on each task is more time-consuming. Excluding tasks that did not generate executable code, Figure 2 shows 200 iterations of OMNI-EPIC with the R2D2 robot. Each node represents a generated task, which includes the natural language task description and environment code. The color of the nodes corresponds to the generation number. For better visualization, each task (natural language description and environment code) is encoded using a pre-trained encoder language model (OpenAI’s text-embedding-3-small (OpenAI, 2024)) and then projected into a 2-dimensional space using t-SNE (Van der Maaten & Hinton, 2008). We manually selected a few tasks that are well-distributed across the embedding space to show a sample of the diversity and creativity of OMNI-EPIC. Connections between nodes indicate which tasks were used as stepping stones to generate the next task, meaning they were provided in context to the task generator. For better readability, Figure 2 only displays the parent task that is closest to the child task in the embedding space and the header text of task descriptions. Appendix D shows the full natural language descriptions of the magnified tasks and provides a more comprehensive visualization of all parent-child task connections. Appendix N shows the three task descriptions that seed the run.

OMNI-EPIC creates tasks that evolve and diverge across the embedding space, forming clusters of related and increasingly complex challenges (Figure 2). For example, the bottom-right region features tasks involving kicking a ball into a goal. Moving towards the bottom-left, the tasks gradually include dynamic elements such as moving obstacles or targets, while still focusing on kicking a ball. As we traverse to the top-left region, the focus shifts to tasks that require pushing or delivering objects into

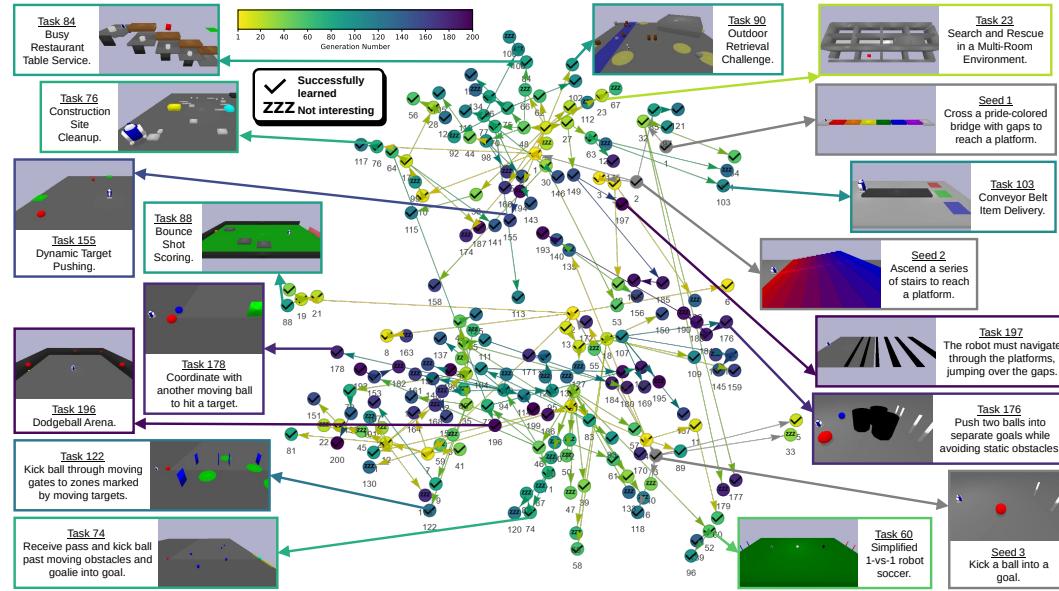


Figure 2: **Long Run with Simulated Learning.** OMNI-EPIC generates a diverse array of tasks, ranging from wildly different objectives to interesting variations of similar overarching tasks. The node color reflects the generation number of the task. A check mark in the node means that the task was successfully learned. A ZZZ symbol means that the task was deemed uninteresting and discarded. The node connections illustrate which tasks were conditioned on when asking an FM to generate a similar yet new and interesting task. Grey nodes show task description seeds that initialized the run.

designated receptacles or target places. This marks a significant departure from the ball-kicking tasks and showcases the diversity of the generated environments. The top-right portion of the space is characterized by generations that emphasize navigation challenges, such as traversing (often moving) platforms or navigating across varied terrains.

Furthermore, there are niches where some generations appear to be variations of each other (Figure 2). For example, in the top-right corner, the generations surrounding task 90 involve retrieving an object and returning it to a designated location. While the overarching objective surrounding that niche remains consistent, the tasks and their learning environments exhibit notable differences. These variations manifest in several aspects, such as the simulated world in which the task takes place (e.g., task 90 is outdoors, task 24 is in connected rooms, task 27 is in two levels connected by a ramp). Other variations include the number of objects that need to be retrieved (e.g., task 102 requires multiple objects to be retrieved, while task 90 only requires one), and the allotted time for completing the task (e.g., task 27 has a time limit of 3 minutes, while task 24 has a time limit of 5 minutes). These task variations can potentially allow for the development of robust and adaptable agents capable of handling a wide range of scenarios (Adaptive Agent Team et al., 2023).

Overall, OMNI-EPIC generates tasks that significantly diverge from the seed tasks used to initialize the archive (the grey nodes in Figure 2). For example, despite the absence of dynamic objects in the seed tasks, OMNI-EPIC generates a substantial number of tasks that incorporate dynamic objects and interactions, such as platforms that move horizontally and vertically, buttons and levers to activate, and moving obstacles. OMNI-EPIC not only explores different task niches (e.g., navigating across different terrains vs. retrieving objects) but also generates interesting variations within each niche (e.g., retrieving objects in different simulated world settings).

## 5 SHORT RUN WITH LEARNING

Previous work (Adaptive Agent Team et al., 2023) has shown that training a large model on a sufficiently diverse set of tasks can produce a generalist agent capable of generalizing to new tasks within that distribution. However, generating such a distribution is both time-consuming and challenging. Furthermore, to achieve efficiency, it is crucial to automatically create a curriculum within this distribution. Since these are the main challenges, our results section focuses on demonstrating that

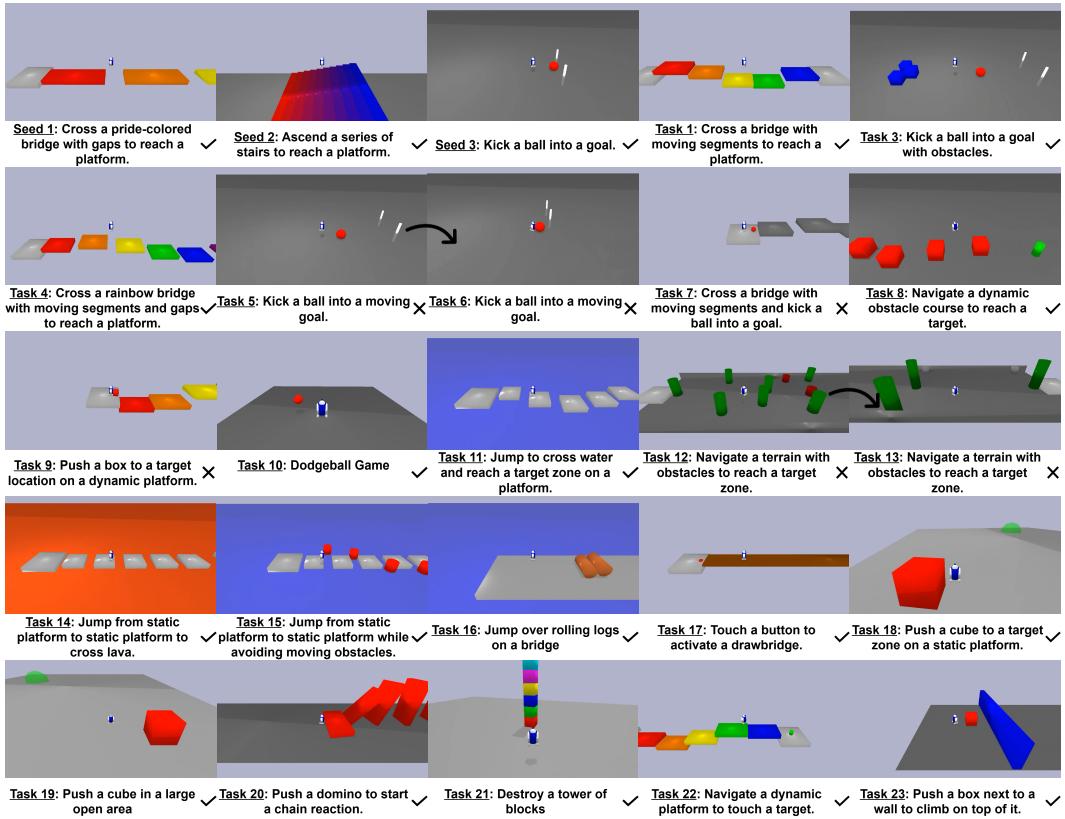


Figure 3: **Short Run with Learning.** OMNI-EPIC adapts to the current capabilities of trained RL agents, generating tasks that are both interesting and learnable. Tasks deemed interesting that are successfully learned are marked by a check and failures by a cross. Uninteresting tasks are not trained on and hence not included here. Arrows between tasks indicate instances where OMNI-EPIC modified a task that the RL agent failed to learn, adjusting the task difficulty to facilitate learning.

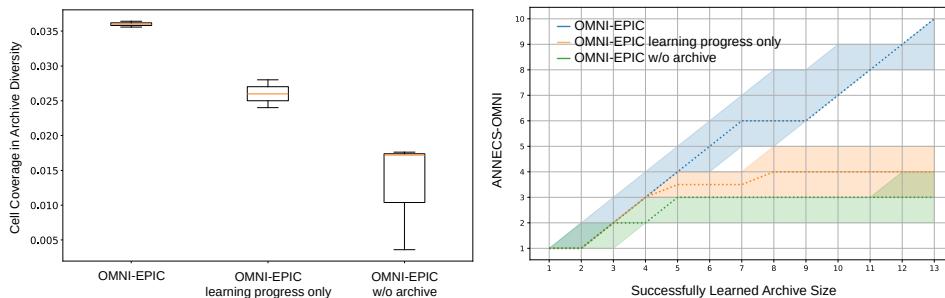
OMNI-EPIC addresses both issues effectively. While our academic lab lacks the resources to train generalist agents on a large set of OMNI-EPIC-generated tasks, our findings suggest that it should be feasible to do so, as OMNI-EPIC generates a progressively complex and diverse curriculum. Nonetheless, we acknowledge that training such agents would still be a significant endeavor, as demonstrated by Adaptive Agent Team et al. (2023), which required an extensive distribution of environments, considerable effort, and substantial computational resources.

To demonstrate OMNI-EPIC’s ability to generate tasks of suitable difficulty for training RL agents, we conducted 5 short runs with RL agent training. Due to limited computational resources in our academic lab, the runs are shorter, but still show the creative potential of OMNI-EPIC and its ability to tailor tasks to the agents’ abilities. The success detector (Section 3.6) evaluates if the agent has successfully completed each task. All short runs are initialized with 3 task description seeds (Appendix N). We find that OMNI-EPIC can effectively generate tasks and environment code that are not only creative and interesting but also learnable and appropriately challenging for RL agents (Figure 3, Appendix F). In the run shown in Figure 3, the RL agents successfully completed 16 tasks, failed at 6 tasks, and 1 task was deemed uninteresting by the post-generation MoI. We conducted a user study with 50 participants and found a 72.7% alignment rate between human evaluations and the success detector’s assessments (Appendix H). Figure 3 shows images of the trained agents and their corresponding tasks, displaying only the header text of the task descriptions for better readability. Appendix E shows the environment code with full task descriptions, images, and a task graph.

OMNI-EPIC leverages previously learned tasks as stepping stones to generate and master more challenging tasks. This iterative process allows RL agents to build upon existing skills to tackle increasingly complex environments. For example, since RL agents in the archive successfully learned to cross a pride-colored bridge with gaps (seed 1) and cross a bridge with moving segments (task 1), OMNI-EPIC branched off these tasks to generate a more challenging task (task 4) combining

these elements. This new task required the agent to cross a rainbow bridge with gaps and moving segments. By continuing training from the policy learned on task 1, the RL agent successfully completed task 4. Furthermore, OMNI-EPIC considers tasks that the RL agents agents failed to learn. For example, since the RL agent failed to learn how to push a box to a target location on a dynamic platform (task 9), future tasks involving crossing platforms did not include pushing objects across them (e.g., tasks 11, 14, 15). This ensures that the generated tasks remain learnable and do not repeatedly incorporate overly difficult challenges. Similarly, when the RL agent failed to navigate through a terrain with obstacles (task 12), OMNI-EPIC generates an easier obstacle course (task 13) that maintains the same objective but reduces the number of obstacles. By using the agent’s past experiences (both successes and failures) as building blocks, OMNI-EPIC generates a curriculum of tasks that progressively and interestingly increases in difficulty while remaining learnable (additional ablations in Appendix G). This adaptive task generation process showcases OMNI-EPIC’s capacity to create a tailored curriculum that maintains an appropriate level of challenge, ensuring that the generated tasks are neither too simple nor too complex.

## 6 QUANTITATIVE RESULTS



**Figure 4: OMNI-EPIC generates significantly more diverse tasks and continues to innovate throughout the run.** (Left) Cell coverage of archive diversity plots in long runs with simulated learning by OMNI-EPIC and the controls. (Right) ANNECS-OMNI measure of progress for OMNI-EPIC and the controls. Dotted lines are median values, shaded regions are 95% confidence intervals.

To evaluate the impact of having a task archive and the contribution of the notion of interestingness in OMNI-EPIC, we compare against two controls: (1) OMNI-EPIC without the task archive (**OMNI-EPIC w/o archive**), and (2) OMNI-EPIC without the models of interestingness (by removing the request for interesting tasks from the task generator prompts and skipping the post-generation MoI step) (**OMNI-EPIC Learning Progress Only**). The quantitative measures for comparisons are described next, and OMNI-EPIC significantly outperforms the controls on both metrics.

**Task Diversity.** To measure the diversity of generated task archives, we plot the cell coverage of a task archive in a 2D discretized plot. First, we use a pretrained text embedding model (OpenAI’s text-embedding-3-small (OpenAI, 2024)) to encode the generated tasks (natural language description and environment code), then reduce the dimensionality to two via PCA (Maćkiewicz & Ratajczak, 1993) (across all tasks from all methods) for easier visualization. We create 2D discretized plots by selecting a discretization level (e.g., 50 in Figure 4) and generating uniform bins across the minimum and maximum values from the PCA embeddings. Each task in the archive is placed in the appropriate bin, and we count the number of unique bins the algorithm fills, a standard measure of diversity (Mouret & Clune, 2015; Pugh et al., 2016). Each method is run with simulated learning, repeated 3 times. OMNI-EPIC achieves significantly higher cell coverage than the controls ( $p < 0.05$ , Mann-Whitney U test). This shows that both the task archive and the MoI significantly and quantifiably contribute to OMNI-EPIC’s ability to generate more diverse tasks (Figure 4). Appendix I shows the archive diversity plots for each method and that the same results hold for different discretization levels.

**Measure of Progress.** While open-ended systems aim to endlessly create and learn new tasks, the question of how to measure progress in such systems remains. Wang et al. (2020) proposed tracking the Accumulated Number of Novel Environments Created and Solved (ANNECS) throughout the run of an open-ended system. Specifically, ANNECS requires that an environment created at a particular iteration (1) is not too hard or easy for the agent to learn and (2) must eventually be solved by the system. However, ANNECS lacks a measure of how interesting or novel the newly generated task is.

To address this limitation, we introduce a new metric to measure progress in open-ended systems, ANNECS-OMNI. Inspired by ideas in Zhang et al. (2023), ANNECS-OMNI adds a third criterion to ANNECS: the new task must be considered interesting compared to previous tasks (approximated here by asking an FM if the task is interesting given the archive of already-solved tasks).

Each method is run with RL training, repeated 5 times. OMNI-EPIC achieves significantly higher ANNECS-OMNI scores than the controls ( $p < 0.05$ , Mann-Whitney U test). As the run proceeds, the ANNECS-OMNI metric consistently increases for OMNI-EPIC, indicating that, for as long as we could afford to run it, the algorithm continuously creates meaningfully new and interesting tasks without stagnation (Figure 4). That is a new high watermark in our field’s longstanding quest to create open-ended algorithms. The ANNECS metric for the different methods can be found in Appendix I.

## 7 DISCUSSION, FUTURE WORK, CONCLUSION

OMNI-EPIC provides a general recipe for generating a potentially endless stream of learnable and interesting environments. By leveraging FMs to generate tasks and their corresponding environment (and reward) code, OMNI-EPIC eliminates the need for handcrafted parameters or predefined task distributions, unshackling algorithms to have the potential to be truly open-ended. OMNI-EPIC can produce a wide variety of tasks, from simple to complex, and allows for the creation of environments that continuously adapt in response to an agent’s developing capabilities. The results demonstrate its effectiveness in generating diverse and creative tasks, highlighting the potential of this approach for training more capable and intelligent agents. By leveraging the generative capabilities of FMs to create a vast array of unique and challenging environments, OMNI-EPIC brings us one step closer to achieving Darwin Completeness, the ability to create *any* possible learning environment.

While we cannot rule out that OMNI-EPIC is creating environments similar to those in its training data, (1) even if it were, this would still be useful for open-endedness; (2) we believe it is generating novel environments, as we were unable to find the same environments when searching online; and (3) with longer runs, we hypothesize it could generate environments endlessly. Its task generator and MoI could generalize well outside the distribution of human data when conditioned on a growing archive of discoveries, though studying this remains a fascinating area for future research.

However, it is important to acknowledge that the current implementation of OMNI-EPIC is not yet Darwin Complete, primarily due to the limitations imposed by the choice of simulator. While OMNI-EPIC can generate a wide variety of tasks and environments, it is constrained by the capabilities and assumptions of the underlying simulation platform. To achieve true Darwin Completeness, OMNI-EPIC could simply be allowed to generate any code, including the ability to download, install, use, or modify any existing simulator, or even write the code for an entirely new simulator from scratch. Since the programming language used here (Python) is Turing Complete, generating code can, in principle, create any computable environment. Enabling the generation of arbitrary code would unlock the full potential of OMNI-EPIC and bring us closer to realizing the vision of Darwin Completeness. Of course, ever smarter FMs are required to take advantage of this opportunity.

The current implementation of OMNI-EPIC trains a population of specialist agents. However, this approach may not fully capitalize on the potential for agents to generalize across a broader range of tasks. Future research could explore alternative training modalities. One promising direction is to train a single policy across all environments (Adaptive Agent Team et al., 2023), which could encourage the development of more versatile and adaptable agents. Another approach is to prioritize environments based on learning progress (Baranes & Oudeyer, 2013; Kanitscheider et al., 2021), focusing on tasks that provide the most opportunities for improvement. Each of these strategies introduces unique dynamics into the open-ended environment generation process, and understanding their effects on agent performance and generalization represents an exciting avenue for future research.

In conclusion, OMNI-EPIC represents a leap towards open-ended learning by generating a potentially endless stream of learnable and interesting tasks. Intriguingly, it also provides a new way of creating human entertainment and educational resources by offering a limitless supply of engaging challenges (Appendix B). OMNI-EPIC could potentially be applied in myriad ways, covering anything from math problems and poetry challenges to games and virtual worlds. By leveraging FMs to create tasks and environment code, OMNI-EPIC opens up a vast space of possibilities for AI and human agents to explore and master. By combining that expressive power with human notions of interestingness, OMNI-EPIC presents a promising path towards the development of truly open-ended and creative AI.

## REFERENCES

- Adaptive Agent Team, Jakob Bauer, Kate Baumli, Satinder Baveja, Feryal Behbahani, Avishkar Bhoopchand, Nathalie Bradley-Schmieg, Michael Chang, Natalie Clay, Adrian Collister, et al. Human-timescale adaptation in an open-ended task space. *arXiv preprint arXiv:2301.07608*, 2023.

Ilge Akkaya, Marcin Andrychowicz, Maciek Chociej, Mateusz Litwin, Bob McGrew, Arthur Petron, Alex Paino, Matthias Plappert, Glenn Powell, Raphael Ribas, et al. Solving rubik's cube with a robot hand. *arXiv preprint arXiv:1910.07113*, 2019.

Adrien Baranes and Pierre-Yves Oudeyer. Active learning of inverse models with intrinsically motivated goal exploration in robots. *Robotics and Autonomous Systems*, 61(1):49–73, 2013.

Yoshua Bengio, Geoffrey Hinton, Andrew Yao, Dawn Song, Pieter Abbeel, Yuval Noah Harari, Ya-Qin Zhang, Lan Xue, Shai Shalev-Shwartz, Gillian Hadfield, et al. Managing AI risks in an era of rapid progress. *arXiv preprint arXiv:2310.17688*, pp. 18, 2023.

Rishi Bommasani, Drew A Hudson, Ehsan Adeli, Russ Altman, Simran Arora, Sydney von Arx, Michael S Bernstein, Jeannette Bohg, Antoine Bosselut, Emma Brunskill, et al. On the opportunities and risks of foundation models. *arXiv preprint arXiv:2108.07258*, 2021.

Herbie Bradley, Andrew Dai, Hannah Teufel, Jenny Zhang, Koen Oostermeijer, Marco Bellagente, Jeff Clune, Kenneth Stanley, Grégory Schott, and Joel Lehman. Quality-diversity through AI feedback. *arXiv preprint arXiv:2310.13032*, 2023.

Tom B Brown. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020.

Jake Bruce, Michael D Dennis, Ashley Edwards, Jack Parker-Holder, Yuge Shi, Edward Hughes, Matthew Lai, Aditi Mavalankar, Richie Steigerwald, Chris Apps, et al. Genie: Generative interactive environments. In *Forty-first International Conference on Machine Learning*, 2024.

Jeff Clune. AI-GAs: AI-generating algorithms, an alternate paradigm for producing general artificial intelligence. *arXiv preprint arXiv:1905.10985*, 2019.

Cédric Colas, Pierre Fournier, Mohamed Chetouani, Olivier Sigaud, and Pierre-Yves Oudeyer. Curious: intrinsically motivated modular multi-goal reinforcement learning. In *International conference on machine learning*, pp. 1331–1340. PMLR, 2019.

Cédric Colas, Tristan Karch, Olivier Sigaud, and Pierre-Yves Oudeyer. Autotelic agents with intrinsically motivated goal-conditioned reinforcement learning: a short survey. *Journal of Artificial Intelligence Research*, 74:1159–1199, 2022.

Erwin Coumans and Yunfei Bai. Pybullet, a python module for physics simulation for games, robotics and machine learning. 2016.

Michael Dennis, Natasha Jaques, Eugene Vinitsky, Alexandre Bayen, Stuart Russell, Andrew Critch, and Sergey Levine. Emergent complexity and zero-shot transfer via unsupervised environment design. *Advances in neural information processing systems*, 33:13049–13061, 2020.

Li Ding, Jenny Zhang, Jeff Clune, Lee Spector, and Joel Lehman. Quality diversity through human feedback. *arXiv preprint arXiv:2310.12103*, 2023.

Yuqing Du, Ksenia Konyushkova, Misha Denil, Akhil Raju, Jessica Landon, Felix Hill, Nando de Freitas, and Serkan Cabi. Vision-language models as success detectors. *arXiv preprint arXiv:2303.07280*, 2023.

Adrien Ecoffet, Jeff Clune, and Joel Lehman. Open questions in creating safe open-ended AI: tensions between control and creativity. In *Artificial Life Conference Proceedings 32*, pp. 27–35. MIT Press One Rogers Street, Cambridge, MA 02142-1209, USA journals-info . . . , 2020.

Danijar Hafner, Jurgis Pasukonis, Jimmy Ba, and Timothy Lillicrap. Mastering diverse domains through world models. *arXiv preprint arXiv:2301.04104*, 2023.

- 594 Nicolas Heess, Dhruva Tb, Srinivasan Sriram, Jay Lemmon, Josh Merel, Greg Wayne, Yuval Tassa,  
 595 Tom Erez, Ziyu Wang, SM Eslami, et al. Emergence of locomotion behaviours in rich environments.  
 596 *arXiv preprint arXiv:1707.02286*, 2017.
- 597
- 598 JE Hopcroft. Introduction to Automata Theory, Languages, and Computation, 2001.
- 599 Andrew Jaegle, Sebastian Borgeaud, Jean-Baptiste Alayrac, Carl Doersch, Catalin Ionescu, David  
 600 Ding, Skanda Koppula, Daniel Zoran, Andrew Brock, Evan Shelhamer, et al. Perceiver io: A  
 601 general architecture for structured inputs & outputs. *arXiv preprint arXiv:2107.14795*, 2021.
- 602
- 603 Minqi Jiang, Edward Grefenstette, and Tim Rocktäschel. Prioritized level replay. In *International  
 Conference on Machine Learning*, pp. 4940–4950. PMLR, 2021.
- 604
- 605 Minqi Jiang, Tim Rocktäschel, and Edward Grefenstette. General intelligence requires rethinking  
 606 exploration. *Royal Society Open Science*, 10(6):230539, 2023.
- 607
- 608 Arthur Juliani, Ahmed Khalifa, Vincent-Pierre Berges, Jonathan Harper, Ervin Teng, Hunter Henry,  
 609 Adam Crespi, Julian Togelius, and Danny Lange. Obstacle tower: A generalization challenge in  
 610 vision, control, and planning. *arXiv preprint arXiv:1902.01378*, 2019.
- 611
- 612 Niels Justesen, Ruben Rodriguez Torrado, Philip Bontrager, Ahmed Khalifa, Julian Togelius, and  
 613 Sebastian Risi. Illuminating generalization in deep reinforcement learning through procedural  
 614 level generation. *arXiv preprint arXiv:1806.10729*, 2018.
- 615
- 616 Ingmar Kanitscheider, Joost Huizinga, David Farhi, William Heben Guss, Brandon Houghton,  
 617 Raul Sampedro, Peter Zhokhov, Bowen Baker, Adrien Ecoffet, Jie Tang, et al. Multi-task cur-  
 618 riculum learning in a complex, visual, hard-exploration domain: Minecraft. *arXiv preprint  
 619 arXiv:2106.14876*, 2021.
- 620
- 621 Victoria Krakovna, Jonathan Uesato, Vladimirand Mikulik, Matthew Rahtz, Tom Everitt,  
 622 Ramana Kumar, Zac Kenton, Jan Leike, and Shane Legg. Specification gaming:  
 623 the flip side of AI ingenuity. [https://deepmind.google/discover/blog/  
 624 specification-gaming-the-flip-side-of-ai-ingenuity/](https://deepmind.google/discover/blog/specification-gaming-the-flip-side-of-ai-ingenuity/), 2020.
- 625
- 626 Joel Lehman, Jonathan Gordon, Shawn Jain, Kamal Ndousse, Cathy Yeh, and Kenneth O Stanley.  
 627 Evolution through large models. In *Handbook of Evolutionary Machine Learning*, pp. 331–366.  
 628 Springer, 2023.
- 629
- 630 Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal,  
 631 Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. Retrieval-augmented genera-  
 632 tion for knowledge-intensive nlp tasks. *Advances in Neural Information Processing Systems*, 33:  
 633 9459–9474, 2020.
- 634
- 635 Bryan Lim, Manon Flageat, and Antoine Cully. Large Language Models as In-context AI Generators  
 636 for Quality-Diversity. *arXiv preprint arXiv:2404.15794*, 2024.
- 637
- 638 Yecheng Jason Ma, William Liang, Guanzhi Wang, De-An Huang, Osbert Bastani, Dinesh Jayaraman,  
 639 Yuke Zhu, Linxi Fan, and Anima Anandkumar. Eureka: Human-level reward design via coding  
 640 large language models. *arXiv preprint arXiv:2310.12931*, 2023.
- 641
- 642 Andrzej Maćkiewicz and Waldemar Ratajczak. Principal components analysis (PCA). *Computers &  
 643 Geosciences*, 19(3):303–342, 1993.
- 644
- 645 Jean-Baptiste Mouret and Jeff Clune. Illuminating search spaces by mapping elites. *arXiv preprint  
 646 arXiv:1504.04909*, 2015.
- 647
- 648 OpenAI. Text embedding 3 small. [https://platform.openai.com/docs/guides/  
 649 embeddings/embedding-models](https://platform.openai.com/docs/guides/embeddings/embedding-models), 2024. Accessed: 17 May 2024.
- 650
- 651 Pierre-Yves Oudeyer and Frederic Kaplan. What is intrinsic motivation? A typology of computational  
 652 approaches. *Frontiers in neurorobotics*, 1:108, 2007.
- 653
- 654 Pierre-Yves Oudeyer, Frdric Kaplan, and Verena V Hafner. Intrinsic motivation systems for au-  
 655 tonomous mental development. *IEEE transactions on evolutionary computation*, 11(2):265–286,  
 656 2007.

- 648 Jack Parker-Holder, Minqi Jiang, Michael Dennis, Mikayel Samvelyan, Jakob Foerster, Edward  
 649 Grefenstette, and Tim Rocktäschel. Evolving curricula with regret-based environment design. In  
 650 *International Conference on Machine Learning*, pp. 17473–17498. PMLR, 2022.
- 651
- 652 Justin K Pugh, Lisa B Soros, and Kenneth O Stanley. Quality diversity: A new frontier for evolutionary  
 653 computation. *Frontiers in Robotics and AI*, 3:202845, 2016.
- 654 Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language  
 655 models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.
- 656
- 657 Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal,  
 658 Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, et al. Learning transferable visual  
 659 models from natural language supervision. In *International conference on machine learning*, pp.  
 660 8748–8763. PMLR, 2021.
- 661
- 662 Mikayel Samvelyan, Akbir Khan, Michael Dennis, Minqi Jiang, Jack Parker-Holder, Jakob Foerster,  
 663 Roberta Raileanu, and Tim Rocktäschel. MAESTRO: Open-ended environment design for multi-  
 664 agent reinforcement learning. *arXiv preprint arXiv:2303.03376*, 2023.
- 665
- Noor Shaker, Julian Togelius, and Mark J Nelson. Procedural content generation in games. 2016.
- 666
- 667 David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche,  
 668 Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering  
 669 the game of Go with deep neural networks and tree search. *nature*, 529(7587):484–489, 2016.
- 670
- Kenneth O Stanley and Joel Lehman. *Why greatness cannot be planned: The myth of the objective*.  
 671 Springer, 2015.
- 672
- Kenneth O Stanley, Joel Lehman, and Lisa Soros. Open-endedness: The last grand challenge you've  
 673 never heard of. *While open-endedness could be a force for discovering intelligence, it could also*  
 674 *be a component of AI itself*, 2017.
- 675
- 676 Shyam Sudhakaran, Miguel González-Duque, Matthias Freiberger, Claire Glanois, Elias Najarro,  
 677 and Sebastian Risi. Mariogpt: Open-ended text2level generation through large language models.  
 678 *Advances in Neural Information Processing Systems*, 36, 2024.
- 679
- 680 Richard S Sutton. Reinforcement learning: An introduction. *A Bradford Book*, 2018.
- 681
- Josh Tobin, Rachel Fong, Alex Ray, Jonas Schneider, Wojciech Zaremba, and Pieter Abbeel. Domain  
 682 randomization for transferring deep neural networks from simulation to the real world. In *2017*  
 683 *IEEE/RSJ international conference on intelligent robots and systems (IROS)*.
- 684
- Graham Todd, Sam Earle, Muhammad Umair Nasir, Michael Cerny Green, and Julian Togelius. Level  
 685 generation through large language models. In *Proceedings of the 18th International Conference on*  
 686 *the Foundations of Digital Games*, pp. 1–8, 2023.
- 687
- Mark Towers, Ariel Kwiatkowski, Jordan Terry, John U Balis, Gianluca De Cola, Tristan Deleu,  
 688 Manuel Goulao, Andreas Kallinteris, Markus Krimmel, Arjun KG, et al. Gymnasium: A standard  
 689 interface for reinforcement learning environments. *arXiv preprint arXiv:2407.17032*, 2024.
- 690
- Laurens Van der Maaten and Geoffrey Hinton. Visualizing data using t-SNE. *Journal of machine*  
 691 *learning research*, 9(11), 2008.
- 692
- A Vaswani. Attention is all you need. *Advances in Neural Information Processing Systems*, 2017.
- 693
- Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and  
 694 Anima Anandkumar. Voyager: An open-ended embodied agent with large language models. *arXiv*  
 695 *preprint arXiv:2305.16291*, 2023a.
- 696
- Lirui Wang, Yiyang Ling, Zhecheng Yuan, Mohit Shridhar, Chen Bao, Yuzhe Qin, Bailin Wang,  
 697 Huazhe Xu, and Xiaolong Wang. Gensim: Generating robotic simulation tasks via large language  
 698 models. *arXiv preprint arXiv:2310.01361*, 2023b.
- 699

702 Rui Wang, Joel Lehman, Jeff Clune, and Kenneth O Stanley. Paired open-ended trailblazer (poet):  
703 Endlessly generating increasingly complex and diverse learning environments and their solutions.  
704 *arXiv preprint arXiv:1901.01753*, 2019.

705  
706 Rui Wang, Joel Lehman, Aditya Rawal, Jiale Zhi, Yulun Li, Jeffrey Clune, and Kenneth Stanley.  
707 Enhanced poet: Open-ended reinforcement learning through unbounded invention of learning  
708 challenges and their solutions. In *International conference on machine learning*, pp. 9940–9951.  
709 PMLR, 2020.

710 Yufei Wang, Zhou Xian, Feng Chen, Tsun-Hsuan Wang, Yian Wang, Katerina Fragkiadaki, Zackory  
711 Erickson, David Held, and Chuang Gan. Robogen: Towards unleashing infinite data for automated  
712 robot learning via generative simulation. *arXiv preprint arXiv:2311.01455*, 2023c.

713 Abhay Zala, Jaemin Cho, Han Lin, Jaehong Yoon, and Mohit Bansal. EnvGen: Generating and Adapt-  
714 ing Environments via LLMs for Training Embodied Agents. *arXiv preprint arXiv:2403.12014*,  
715 2024.

716 Jenny Zhang, Joel Lehman, Kenneth Stanley, and Jeff Clune. Omni: Open-endedness via models of  
717 human notions of interestingness. *arXiv preprint arXiv:2306.01711*, 2023.

719

720

721

722

723

724

725

726

727

728

729

730

731

732

733

734

735

736

737

738

739

740

741

742

743

744

745

746

747

748

749

750

751

752

753

754

755

756    **A REPRODUCIBILITY**  
 757

758    For full transparency and to ensure reproducibility, we will open-source all code associated with this  
 759    work.  
 760

761    **B GAME INTERFACE**  
 762



788    **Figure 5: OMNI-EPIC in a game interface.**  
 789

790    Apart from training agents, OMNI-EPIC can also be a good form of human entertainment! We  
 791    created a game interface whereby players can control the R2D2 robot with keyboard inputs, and play  
 792    the generated tasks (Figure 5). OMNI-EPIC dynamically generates the next interesting level for the  
 793    player, adjusting based on the player's skill level by suggesting tasks that are not too easy or difficult.  
 794    OMNI-EPIC opens a new era for games, where procedural content is automatically generated and  
 795    tailored to the player's abilities, ensuring a consistently engaging experience.  
 796  
 797  
 798  
 799  
 800  
 801  
 802  
 803  
 804  
 805  
 806  
 807  
 808  
 809

810           **C VLM AS SUCCESS DETECTOR**  
 811

812         The success-checking function can be easily implemented in simulated environments where information  
 813         can be readily accessed through code (Section 3.6). However, it may face challenges in  
 814         real-world scenarios, off-the-shelf closed-source video games, or even within simulated environments  
 815         for tasks that involve visual assessment (e.g., building a castle, arranging boxes to resemble an  
 816         elephant), where the required information is not directly available or difficult to evaluate using only  
 817         code. A natural solution is to use VLMs as success detectors (Radford et al., 2021). VLMs can  
 818         potentially detect success on a wider range of tasks (e.g., tidying a room, doing a backflip) (Du et al.,  
 819         2023) than code generation. We input snapshots of the agent’s behavior every second, the natural  
 820         language task description, and environment code (see below). Since our preliminary testing found  
 821         that current VLMs are not yet accurate enough to be used as success detectors, we use code generated  
 822         by LLMs for this purpose instead. However, we expect VLM capabilities to rapidly improve over  
 823         time, eventually achieving higher accuracy and making them viable for future use.  
 824

**System Prompt:**

825         You are an expert in Python programming and reinforcement learning. Your  
 826         goal is to evaluate if a robot has solved a task. You will be  
 827         provided with the task description, the corresponding environment  
 828         code and an image containing snapshots of the robot attempting to  
 829         complete the task. Your objective is to describe the image, reason  
 830         about whether the task has been completed and determine if the robot  
 831         has solved the task.

**Instructions:**

- In the description of the image, describe the environment and the behavior of the robot.
- In the reasoning, analyze if the environment corresponds to the task description and if the behavior of the robot meets the requirements for task success.
- The task is considered failed if the environment is constructed in a way that makes solving the task impossible.
- If you are unsure, make an educated guess and always provide an answer.
- If you are unsure, say that it has failed.

**Robot description:**

{ROBOT\_DESC}

**Desired format:**

Description of the image:  
 <image description>

**Reasoning for task success/failure:**

<reasoning>

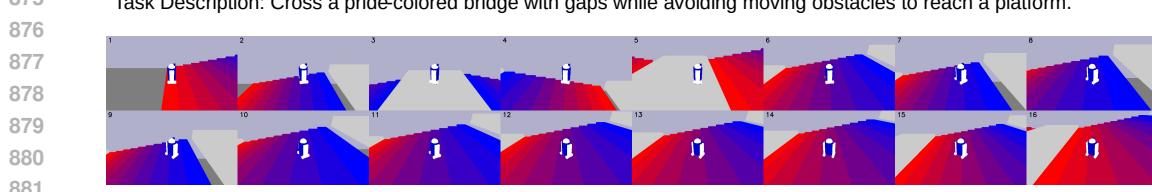
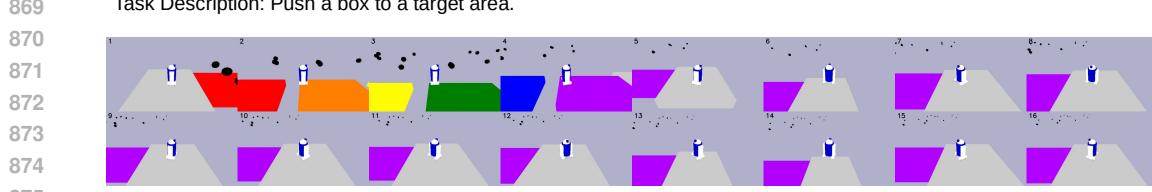
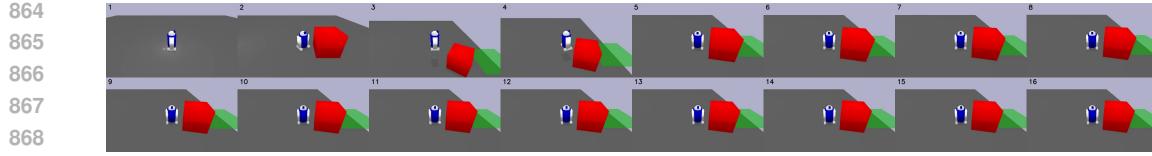
**Did the robot solve the task?:**

<Yes/No>

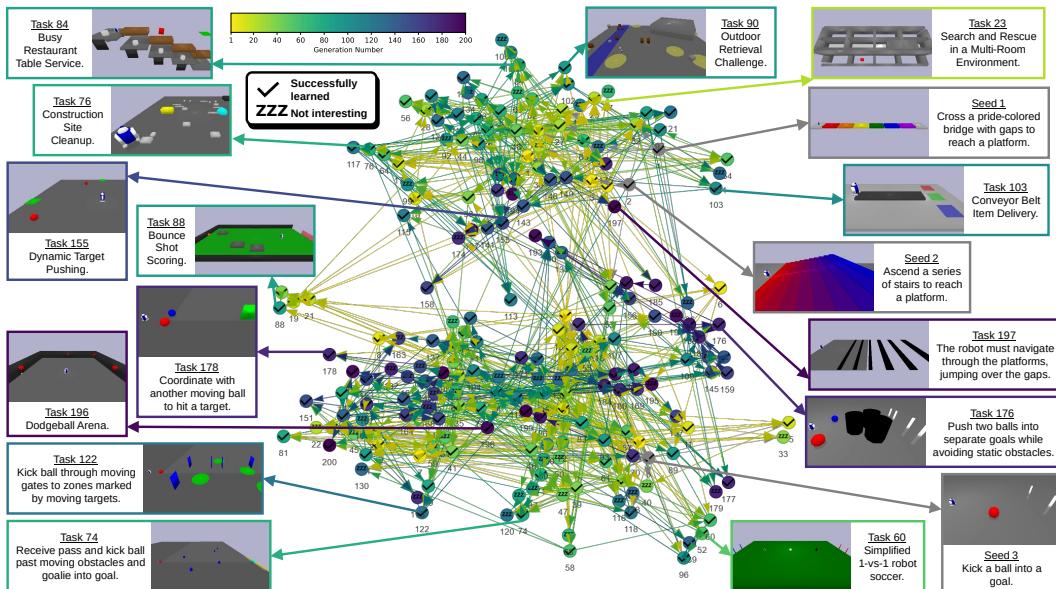
**User Prompt:**

854         Task description and environment code:  
 855         {ENV\_CODE}

**Image Examples:**



## D SUPPLEMENTARY MATERIALS FOR LONG RUN WITH SIMULATED LEARNING



**Figure 6: Long run with simulated learning task graph with all parent-child task connections.**  
This figure presents the same task graph as Figure 2, but with all parent-child task connections displayed. The node color reflects the generation number of the task. A check mark in the node means that the task was successfully learned. A ZZZ symbol means that the task was deemed uninteresting and discarded. The node connections illustrate which tasks were conditioned on when asking an FM to generate a similar yet new and interesting task. Due to the high density of tasks and connections, visualizing all relationships clearly in a static image is challenging. To better understand and navigate the intricate web of task relationships, an interactive version of the task graph is available at <https://dub.sh/omni-epic>.

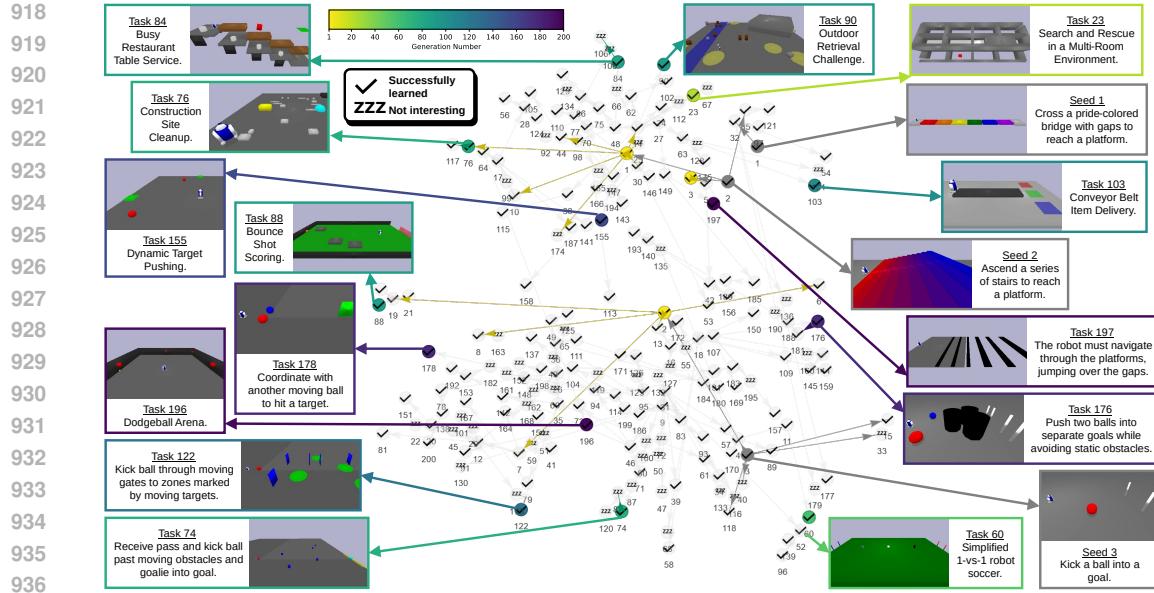


Figure 7: **Long run with simulated learning task graph where only the nodes with enlarged task images are colored.** This figure presents the same task graph as Figure 2, but all nodes are grayed out except for those with enlarged task images, which are colored.

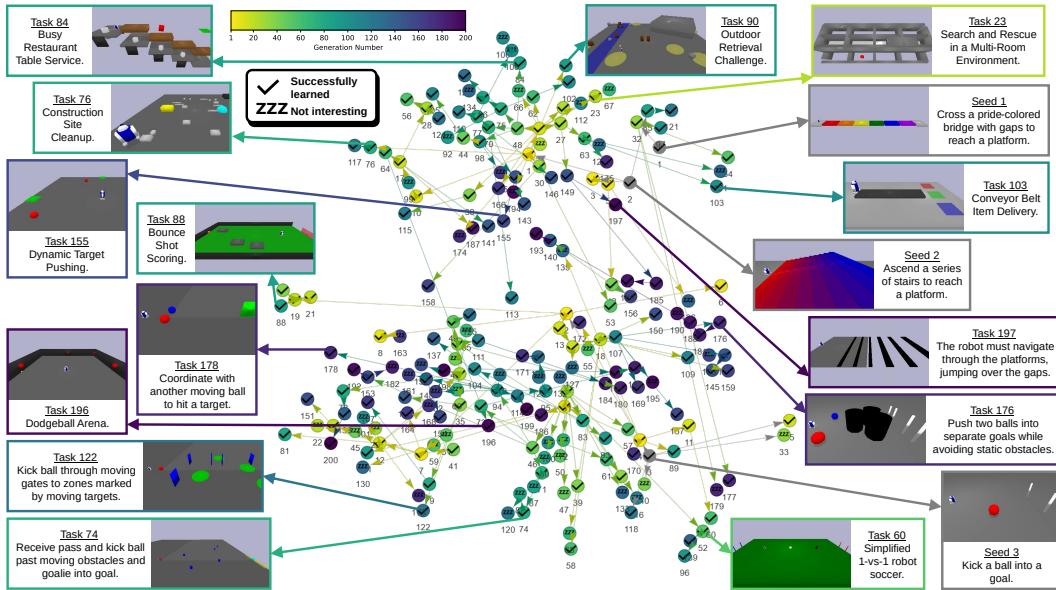


Figure 8: **Long run with simulated learning task graph with thinner visualization edges.** This figure presents the same task graph as Figure 2, but the edges are visualized with reduced thickness.

The natural language descriptions and environment code for all tasks shown in Figure 2 are available at <https://dub.sh/omni-epic>. For conciseness, we present the full natural language descriptions of all magnified tasks, and the environment code of one magnified task (Task 76):

```

import numpy as np
from oped.envs.r2d2.base import R2D2Env

class Env(R2D2Env):
    """

```

```

972     Construction Site Cleanup
973
974     Description:
975     - The environment is a construction site with dimensions 10 m x 10 m.
976     - There are 3 piles of different types of debris scattered around the
977         construction site:
978         - 5 bricks (each 0.2 m x 0.2 m x 0.2 m)
979         - 5 metal scraps (each 0.5 m x 0.25 m x 0.25 m)
980         - 5 wooden planks (each 1 m x 0.2 m x 0.1 m)
981     - There are also 2 robotic construction vehicles moving around the
982         site in pre-programmed patterns:
983         - A bulldozer that pushes dirt piles around in a rectangle, 4 m x 3
984             m
985         - An excavator that swings its arm and bucket back and forth in a
986             180 degree arc, 2 m in radius
987     - On one side of the construction site are 3 square receptacle bins,
988         each 1 m x 1 m x 0.5 m tall, labeled for each type of debris (
989             bricks, metal, wood).
990     - The robot starts in one corner of the site.
991
992     Task:
993     The robot needs to pick up each piece of debris and place it in the
994         correct receptacle, while avoiding collisions with the moving
995         construction vehicles. The robot can only carry one piece of
996         debris at a time.
997
998     Success Conditions:
999     The task is complete when all pieces of debris have been placed in
1000         their correct bins, and the robot has returned to its starting
1001         position.
1002
1003     Time Limit:
1004     The robot has 10 minutes to complete the cleanup task.
1005
1006     Rewards:
1007     - Provide a small reward for each piece of debris successfully picked
1008         up.
1009     - Provide a moderate reward for each piece of debris placed in the
1010         correct bin.
1011     - Provide a large reward for completing the task and returning to the
1012         start position.
1013     - Provide a small penalty for each collision with the construction
1014         vehicles.
1015
1016     Termination:
1017     The episode ends if the robot flips over, or if the time limit is
1018         exceeded.
1019     """
1020
1021     def __init__(self):
1022         super().__init__()
1023         self.site_size = [10.0, 10.0, 0.1]
1024         self.site_position = [0.0, 0.0, 0.0]
1025         self.site_id = self.create_box(mass=0.0, half_extents=[self.
1026             site_size[0] / 2, self.site_size[1] / 2, self.site_size[2] /
1027             2], position=self.site_position, color=[0.5, 0.5, 0.5, 1.0])
1028         self._p.changeDynamics(bodyUniqueId=self.site_id, linkIndex=-1,
1029             lateralFriction=0.8, restitution=0.5)
1030         self.brick_size = [0.2, 0.2, 0.2]
1031         self.metal_size = [0.5, 0.25, 0.25]
1032         self.wood_size = [1.0, 0.2, 0.1]
1033         self.debris_sizes = [self.brick_size, self.metal_size, self.
1034             wood_size]
1035         self.num_debris_types = len(self.debris_sizes)
1036         self.num_debris_each = 5

```

```

1026     self.debris_ids = []
1027     for i in range(self.num_debris_types):
1028         for _ in range(self.num_debris_each):
1029             debris_position = [np.random.uniform(-self.site_size[0] / 2 + 2.0, self.site_size[0] / 2 - 2.0), np.random.uniform(-self.site_size[1] / 2 + 2.0, self.site_size[1] / 2 - 2.0), self.debris_sizes[i][2] / 2]
1030             debris_id = self.create_box(mass=1.0, half_extents=[self.debris_sizes[i][0] / 2, self.debris_sizes[i][1] / 2, self.debris_sizes[i][2] / 2], position=debris_position, color=[0.8, 0.8, 0.8, 1.0])
1031             self.debris_ids.append((debris_id, i))
1032     self.bulldozer_size = [1.0, 0.5, 0.5]
1033     self.bulldozer_position_init = [-self.site_size[0] / 4, 0.0, self.bulldozer_size[2] / 2]
1034     self.bulldozer_id = self.create_box(mass=0.0, half_extents=[self.bulldozer_size[0] / 2, self.bulldozer_size[1] / 2, self.bulldozer_size[2] / 2], position=self.bulldozer_position_init, color=[1.0, 1.0, 0.0, 1.0])
1035     self.excavator_radius = 2.0
1036     self.excavator_position_init = [self.site_size[0] / 4, 0.0, 0.0]
1037     self.excavator_id = self.create_sphere(mass=0.0, radius=0.5, position=self.excavator_position_init, color=[0.0, 1.0, 1.0, 1.0])
1038     self.receptacle_size = [1.0, 1.0, 0.5]
1039     self.receptacle_positions = [[self.site_size[0] / 2 - self.receptacle_size[0] / 2, -self.site_size[1] / 3, self.receptacle_size[2] / 2], [self.site_size[0] / 2 - self.receptacle_size[0] / 2, 0.0, self.receptacle_size[2] / 2], [self.site_size[0] / 2 - self.receptacle_size[0] / 2, self.site_size[1] / 3, self.receptacle_size[2] / 2]]
1040     self.receptacle_ids = []
1041     for i in range(self.num_debris_types):
1042         receptacle_id = self.create_box(mass=0.0, half_extents=[self.receptacle_size[0] / 2, self.receptacle_size[1] / 2, self.receptacle_size[2] / 2], position=self.receptacle_positions[i], color=[0.2, 0.2, 0.2, 1.0])
1043         self.receptacle_ids.append(receptacle_id)
1044     self.robot_position_init = [-self.site_size[0] / 2 + 2.0, -self.site_size[1] / 2 + 2.0, self.site_size[2] + self.robot.links['base'].position_init[2] + 0.1]
1045     self.robot_orientation_init = self._p.getQuaternionFromEuler([0.0, 0.0, np.pi / 4])
1046     self.time_limit = 600.0
1047     self.debris_pick_reward = 1.0
1048     self.debris_place_reward = 10.0
1049     self.task_complete_reward = 100.0
1050     self.collision_penalty = -1.0
1051
1052     def create_box(self, mass, half_extents, position, color):
1053         collision_shape_id = self._p.createCollisionShape(shapeType=self._p.GEOM_BOX, halfExtents=half_extents)
1054         visual_shape_id = self._p.createVisualShape(shapeType=self._p.GEOM_BOX, halfExtents=half_extents, rgbaColor=color)
1055         return self._p.createMultiBody(baseMass=mass,
1056                                         baseCollisionShapeIndex=collision_shape_id,
1057                                         baseVisualShapeIndex=visual_shape_id, basePosition=position)
1058
1059     def create_sphere(self, mass, radius, position, color):
1060         collision_shape_id = self._p.createCollisionShape(shapeType=self._p.GEOM_SPHERE, radius=radius)
1061         visual_shape_id = self._p.createVisualShape(shapeType=self._p.GEOM_SPHERE, radius=radius, rgbaColor=color)
1062
1063
1064
1065
1066
1067
1068
1069
1070
1071
1072
1073
1074
1075
1076
1077
1078
1079

```

```

1080     return self._p.createMultiBody(baseMass=mass,
1081         baseCollisionShapeIndex=collision_shape_id,
1082         baseVisualShapeIndex=visual_shape_id, basePosition=position)
1083
1084     def get_object_position(self, object_id):
1085         return np.asarray(self._p.getBasePositionAndOrientation(object_id
1086                         )[0])
1087
1088     def reset(self):
1089         observation = super().reset()
1090         self.time = 0.0
1091         self.debris_picked = [False] * len(self.debris_ids)
1092         self.debris_placed = [False] * len(self.debris_ids)
1093         self._p.resetBasePositionAndOrientation(self.robot.robot_id, self
1094             .robot_position_init, self.robot_orientation_init)
1095         return observation
1096
1097     def step(self, action):
1098         observation, reward, terminated, truncated, info = super().step(
1099             action)
1100         self.time += self.dt
1101         bulldozer_position = self.get_object_position(self.bulldozer_id)
1102         bulldozer_position[0] = self.bulldozer_position_init[0] + 2.0 *
1103             np.sin(2 * np.pi * self.time / 20.0)
1104         bulldozer_position[1] = self.bulldozer_position_init[1] + 1.5 *
1105             np.sin(2 * np.pi * self.time / 30.0)
1106         self._p.resetBasePositionAndOrientation(self.bulldozer_id,
1107             bulldozer_position, [0.0, 0.0, 0.0, 1.0])
1108         excavator_position = self.get_object_position(self.excavator_id)
1109         excavator_position[0] = self.excavator_position_init[0] + self.
1110             excavator_radius * np.cos(np.pi * self.time / 10.0)
1111         excavator_position[1] = self.excavator_position_init[1] + self.
1112             excavator_radius * np.sin(np.pi * self.time / 10.0)
1113         self._p.resetBasePositionAndOrientation(self.excavator_id,
1114             excavator_position, [0.0, 0.0, 0.0, 1.0])
1115         return (observation, reward, terminated, truncated, info)
1116
1117     def get_task_rewards(self, action):
1118         reward_pick = 0.0
1119         reward_place = 0.0
1120         reward_complete = 0.0
1121         penalty_collision = 0.0
1122         for i, (debris_id, debris_type) in enumerate(self.debris_ids):
1123             if not self.debris_picked[i] and len(self._p.getContactPoints
1124                 (bodyA=self.robot.robot_id, bodyB=debris_id)) > 0:
1125                 self.debris_picked[i] = True
1126                 reward_pick += self.debris_pick_reward
1127             if self.debris_picked[i] and (not self.debris_placed[i]) and
1128                 (len(self._p.getContactPoints(bodyA=debris_id, bodyB=self
1129                     .receptacle_ids[debris_type])) > 0):
1130                 self.debris_placed[i] = True
1131                 reward_place += self.debris_place_reward
1132             if all(self.debris_placed) and np.linalg.norm(self.robot.links['
1133                 base'].position[:2] - np.asarray(self.robot_position_init
1134                     [:2])) < 1.0:
1135                 reward_complete = self.task_complete_reward
1136             if len(self._p.getContactPoints(bodyA=self.robot.robot_id, bodyB=
1137                 self.bulldozer_id)) > 0 or len(self._p.getContactPoints(bodyA=
1138                     self.robot.robot_id, bodyB=self.excavator_id)) > 0:
1139                 penalty_collision = self.collision_penalty
1140         return {'reward_pick': reward_pick, 'reward_place': reward_place,
1141             'reward_complete': reward_complete, 'penalty_collision':
1142                 penalty_collision}
1143
1144     def get_terminated(self, action):

```

```
1134     if self.time >= self.time_limit:
1135         return True
1136     if np.dot(np.asarray([0, 0, 1]), np.asarray(self._p.
1137         getMatrixFromQuaternion(self.robot.links['base'].orientation)
1138         ).reshape(3, 3)[:, 2]) < 0.5:
1139         return True
1140     return False
1141
1142     def get_success(self):
1143         return all(self.debris_placed) and np.linalg.norm(self.robot.
1144             links['base'].position[:2] - np.asarray(self.
1145                 robot_position_init[:2])) < 1.0
```

### Task 23

Search and Rescue in a Multi-Room Environment

**Description:**

- The environment consists of a large, two-story building with multiple rooms connected by doors, hallways, and staircases. The layout is fixed but contains challenging features:
  - 8 rooms total (4 on each floor), connected by 2 hallways (one per floor)
  - Each room is 5 m x 5 m. Hallways are 20 m long and 2 m wide. Staircase is 5 m long.
  - Rooms contain various static and movable obstacles like furniture.
  - Doors are 1 m wide and initially closed. Robot must push them open.
  - A target object (a 0.5 m x 0.5 m x 1 m box) representing a person is located in one of the rooms on the second floor. Its location is fixed for now.
  - The robot starts at the entrance of the building on the first floor.
  - The robot must systematically explore the building, locate the target object, attach a virtual tether to it, and guide it back to the entrance while avoiding obstacles.

#### **Success Conditions:**

The task is considered complete when the robot brings the target object to rest within 2 m of the entrance location.

Time limit:

The robot has a time limit of 10 minutes to complete the task.

### Rewards:

- Provide a small reward for exploring new areas of the environment.
  - Provide a moderate reward for locating the target object.
  - Provide a large reward for successfully bringing the target object to the goal location.
  - Provide a small penalty for collisions with obstacles to encourage careful navigation.

### Termination:

The episode terminates if the robot flips over or if the time limit is exceeded.

## Task 60

Simplified 1-vs-1 robot soccer.

#### Description:

- The environment consists of a large flat ground representing a soccer field.
  - Two goals are placed on opposite ends of the field, each defined by two posts 2 meters high and 3 meters apart.
  - The robot is placed on one side of the field, facing the opponent's goal.

1188 – A basic opponent robot is placed on the other side, initially  
 1189 stationary.  
 1190 – A soccer ball is placed in the center of the field.  
 1191 – The objective is for the robot to navigate to the ball, take possession  
 1192 of it, and kick it into the opponent's goal while defending its own  
 1193 goal.  
 1194 – In the first stage, the opponent robot remains stationary. In the  
 1195 second stage, the opponent robot also moves to chase after the ball  
 1196 and attempt to kick it into the robot's goal.  
 1197 Success:  
 1198 The task is completed successfully if the robot is able to kick the ball  
 1199 into the opponent's goal while preventing the opponent from scoring.  
 1200 Rewards:  
 1201 – The robot is rewarded for possessing the ball, defined as being within  
 1202 1 meter of the ball while the opponent is not.  
 1203 – The robot is rewarded for bringing the ball closer to the opponent's  
 1204 goal.  
 1205 – The robot is rewarded for kicking the ball with a velocity toward the  
 1206 opponent's goal.  
 1207 – The robot is penalized if the opponent takes possession of the ball or  
 1208 if the ball goes out of bounds.  
 1209 – The robot is greatly rewarded for scoring a goal and penalized if the  
 1210 opponent scores.  
 1211 Termination:  
 1212 The task is terminated if a goal is scored by either side, if the ball  
 1213 goes out of bounds, or if a time limit is reached.

#### Task 74

1214 Receive pass and kick ball past moving obstacles and goalie into goal.  
 1215  
 1216 The robot is placed on a large flat ground designed to look like a soccer  
 1217 /football pitch with markings.  
 1218  
 1219 A teammate robot is positioned 5 meters to the side of the main robot.  
 1220 After a fixed delay at the start of the episode, the teammate robot  
 1221 passes a ball towards a point 3 meters in front of the main robot at  
 1222 a speed of 3 m/s. The pass target point is fixed, but the main robot'  
 1223 s initial position has some randomness, so it will need to adjust to  
 1224 align itself with the passed ball.  
 1225  
 1226 3 cylindrical obstacles of 0.5 meter radius and 1 meter height are placed  
 1227 at random positions 3 to 8 meters in front of the robot's initial  
 1228 position. The obstacles are colored to resemble soccer players and  
 1229 move randomly back and forth over a 2 meter range at speeds between  
 1230 0.5 and 1.5 m/s. The obstacles do not actively chase the ball but  
 1231 will block it if it comes near them.  
 1232  
 1233 A goal area 5 meters wide and 2 meters deep is placed 10 meters in front  
 1234 of the robot's initial position, centered along the front line.  
 1235  
 1236 A goalie obstacle 1 meter wide, 0.5 meters deep, and 1.5 meters tall  
 1237 moves side to side across the goal line at a speed of 1.5 m/s,  
 1238 tracking the ball when it is within 3 meters. The goalie is colored  
 1239 differently than the field obstacles.  
 1240  
 1241 The task is for the main robot to receive the teammate's pass and then  
 1242 kick the moving ball past the field obstacles and goalie to score a  
 1243 goal, without touching any of the obstacles.

#### Task 84

1242

**Busy Restaurant Table Service**

1243

**Description:**

1244

- The environment is a busy restaurant dining area, consisting of a 12 m x 12 m room with 4 tables, an entrance door, and a kitchen door.
- Restaurant patrons (simulated as cylinders) continuously enter from the entrance, move to a table, stay for a period of time, and then exit, making the environment dynamic.
- The robot must perform two types of table service tasks:
  - 1) Table setting: The robot must collect clean dishes and utensils from the kitchen and arrange them properly on the tables in preparation for new patrons. Dishes should be stacked neatly.
  - 2) Table busing: After patrons leave, the robot must clear dirty dishes and utensils from the tables, load them into a bin, and return the bin to the kitchen. The bin has a maximum capacity that the robot must respect.
- The robot must avoid colliding with patrons and restaurant furniture as it navigates.
- Dishes will break if dropped. The robot must handle them carefully.

1245

**Success Conditions:**

1246

- For table setting, success means all necessary dishes and utensils are properly placed on empty tables prior to new patrons arriving. Dishes should be undamaged and neatly stacked.
- For table busing, success involves clearing all dirty dishes from unoccupied tables, loading them into the bin without exceeding capacity, and delivering the bin to the kitchen. No dishes should be broken.
- The robot should avoid any collisions with patrons or furniture throughout the episode.

1247

**Time Limit:**

1248

The episode lasts for 10 minutes of simulated time. The robot must continuously perform both table setting and busing tasks during this period as needed.

1249

**Rewards:**

1250

- Provide a moderate reward for each successful table set, with all dishes placed neatly and undamaged.
- Provide a moderate reward for each successful table busing, with no broken dishes and the bin delivered to the kitchen.
- Give a small reward for carefully handling dishes without breaking them.
- Assign a substantial penalty for any collisions between the robot and patrons or furniture.
- Assign a small penalty for dropping and breaking a dish.
- Provide a small reward at each timestep for maintaining a tidy dining room, with no dirty dishes left on unoccupied tables.

1251

**Termination:**

1252

The episode ends if the robot crashes into a patron or furniture, or if the time limit is reached. The episode is considered successful if the robot consistently performs both table setting and busing throughout the full time period with no collisions or broken dishes.

1253

**Task 88**

1254

**Bounce Shot Scoring Challenge**

1255

**Description:**

1256

- The environment consists of a walled field measuring 20 m x 10 m. The field contains 3 ramps/mounds (triangular prisms) placed randomly, each measuring 2 m long x 1 m wide x 0.5 m high.

- 1296 – A single goal 3 m wide and 1 m high is placed centered on one end of  
 1297 the field.  
 1298 – A ball (0.5 m diameter) is released from a random point along the  
 1299 opposite side of the field from the goal. The ball is given an  
 1300 initial velocity of 3–5 m/s at a random angle towards the goal end of  
 1301 the field.  
 1302 – Due to the ramps on the field, the ball will bounce and roll  
 1303 unpredictably as it crosses the field.  
 1304 – The robot begins in the center of the field on the goal side.

1305 **Task:**

1306 The robot must intercept the bouncing ball and kick it into the goal to  
 1307 score a point. The robot should dynamically adjust its approach to  
 1308 the ball based on the ball's changing trajectory after each bounce  
 1309 off a ramp. The robot needs to time its kick to hit the ball at the  
 1310 optimal point in its bounce to redirect it into the goal.

1311 **Success Conditions:**

1312 The task is completed successfully if the robot causes the ball to fully  
 1313 enter the goal area.

1314 **Rewards:**

- 1315 – Small reward for decreasing distance to the ball, scaled higher as the  
 1316 ball gets closer to the robot's side of the field  
 1317 – Moderate penalty if the ball's velocity drops to zero or it leaves the  
 1318 field without scoring  
 1319 – Large reward for kicking the ball, scaled by the ball's post-kick  
 1320 velocity in the direction of the goal  
 1321 – Large reward for scoring a goal

1322 **Termination:**

1323 The episode ends if the ball comes to rest, leaves the field, or enters  
 1324 the goal area.

1325 **Task 90**

1326 **Outdoor Retrieval Challenge**

1327 **Description:**

- 1328 – The environment is a large outdoor field 50 m x 50 m, with the robot  
 1329 starting in the center.  
 1330 – The terrain includes various features:  
 1331 – A steep hill with a 30 degree incline  
 1332 – A shallow stream, 0.5 m deep and 5 m wide  
 1333 – Patches of loose sand and gravel  
 1334 – Scattered trees and boulders up to 2 m tall  
 1335 – A 0.5 m diameter soccer ball is placed randomly between 10 m and 20 m  
 1336 from the robot's starting position.  
 1337 – The robot must locate the ball, pick it up or get it balanced on its  
 1338 back, and return it to the starting location.  
 1339 – The ball is heavy enough that it can't simply be pushed the whole way.  
 1340 The robot needs to lift/carry it.  
 1341 – If the robot drops the ball or it rolls more than 5 m away, it is  
 1342 replaced at a new random location.

1343 **Success Conditions:**

1344 The task is completed when the robot returns to within 2 m of its  
 1345 starting position with the ball balanced on its back or in its  
 1346 possession.

1347 **Time Limit:**

1348 The robot has 10 minutes to complete the retrieval.

1349 **Rewards:**

- 1350     – Provide a small reward for exploring the environment and locating the  
 1351       ball  
 1352     – Provide a moderate reward for successfully getting the ball balanced on  
 1353       the robot's back or lifted off the ground  
 1354     – Provide a large reward for returning to the start area with the ball  
 1355     – Provide a small penalty if the ball is dropped or lost, to encourage  
 1356       careful handling

1357 **Termination:**

1358 The episode ends if the robot flips over and can't right itself, or if  
 1359       the time limit is exceeded.

1360 **Task 103**

1362 **Conveyor Belt Item Delivery**

1363 **Description:**

- 1364     – The environment consists of two 5 m x 5 m platforms connected by a 5 m  
 1365       long conveyor belt. The conveyor belt moves from left to right at a  
 1366       speed of 0.5 m/s.
- 1367     – On the right platform, there are three target areas marked by 1 m x 1 m  
 1368       colored squares (red, green, blue).
- 1369     – Also on the right platform, there is a dispenser that releases items  
 1370       onto the conveyor belt at random intervals between 2–5 seconds. The  
 1371       items are 0.5 m cubes colored either red, green, or blue.
- 1372     – The robot begins on the left platform.

1373 **Task:**

1374 The robot must jump onto the conveyor belt, pick up the colored items,  
 1375       ride the conveyor to the right platform, and deliver each item to the  
 1376       target area matching its color. The robot should try to deliver as  
 1377       many items as possible to the correct targets within the time limit.

1378 After delivering an item, the robot must jump back onto the conveyor belt  
 1379       and return to the left platform before the next item is released.  
 1380       The robot should wait on the left platform for the next item.

1381 **Rewards:**

- 1382     – Provide a small reward for picking up an item from the conveyor belt.
- 1383     – Provide a large reward for delivering an item to the correct target  
 1384       area based on color.
- 1385     – Provide a small penalty for delivering an item to the wrong target area
- 1386     – Provide a moderate penalty if the robot falls off the conveyor belt or  
 1387       platforms.

1388 **Success Conditions:**

1389 The task is considered complete if the robot successfully delivers at  
 1390       least 5 items to their correct target areas within the time limit.

1392 **Time Limit:**

1393 The robot has 3 minutes to deliver as many items as possible.

1394 **Termination:**

1395 The episode ends if the robot falls off the platforms or conveyor belt,  
 1396       or if the time limit is reached.

1398 **Task 122**

1399 **Kick ball through moving gates to zones marked by moving targets.**

1401 The environment consists of a large flat ground. A ball is placed 5  
 1402       meters directly in front of the robot. After a delay of 1–2 seconds (randomized),  
 1403       the ball begins rolling straight ahead at a constant velocity of 1–2 m/s (randomized).

1404  
 1405 Five vertical rectangular gates, each 3 meters tall and 1 meter wide, are  
 1406 placed at randomized positions between 5 to 25 meters in front of  
 1407 the ball's initial position, at randomized lateral offsets up to 5  
 1408 meters on either side. The gates translate laterally with randomized  
 1409 constant velocities between 0.2–1 m/s, reversing direction each time  
 1410 they move 5 meters from their initial position. The gates' movement  
 1411 is triggered at the start of the episode.  
 1412  
 1413 Five target zones are marked on the ground at randomized positions  
 1414 between 10 to 30 meters in front of the ball's initial position, at  
 1415 randomized lateral offsets up to 10 meters on either side. Each  
 1416 target zone is a circle with radius 2 meters. At the center of each  
 1417 target zone is a flat cylindrical marker, 2 meters in radius and 0.1  
 1418 meters tall. The markers rotate in place with randomized constant  
 1419 angular velocities between 0.1–0.5 rad/s, switching between clockwise  
 1420 and counterclockwise rotation each time the ball passes through a  
 1421 gate. The markers' rotation is triggered at the start of the episode.  
 1422  
 1423 The robot is initialized 3 meters behind the ball, facing the direction  
 1424 of the ball's initial velocity.  
 1425  
 1426 The task is for the robot to kick the ball through each gate in order,  
 1427 aiming to get the ball to stop within the corresponding target zone  
 1428 after it passes through the gate. The robot should decide when and  
 1429 how to kick the ball based on the observed motion of the gates and  
 1430 markers.  
 1431  
 1432 Passing through a gate awards 2 points if the ball subsequently stops in  
 1433 the correct target zone, and 1 point otherwise. The episode ends when  
 1434 the ball passes through all gates, or a maximum time limit is  
 1435 reached. The robot should aim to maximize its total score.

### Task 155

#### Task: Dynamic Target Pushing Challenge

##### Description:

- The environment is a 15 m x 15 m open area with a flat surface.
- There are 3 dynamic targets (spheres with a 0.5 m diameter) that move along predefined linear paths. Each target moves back and forth along a 5 m path at a speed of 0.5 m/s.
- The robot starts at the center of the area.
- Three goal zones (1 m x 1 m squares) are located at the edges of the area: one at the north edge, one at the east edge, and one at the west edge.

##### Task:

The robot must push each moving target into its corresponding goal zone. The targets can be pushed in any order, but each target must come to rest fully within its goal zone to be considered successfully delivered.

##### Initial Positions:

- Robot: Center of the area at (0, 0, 0.5)
- Target 1: Moving along the path from (-2.5, 5, 0.25) to (2.5, 5, 0.25)
- Target 2: Moving along the path from (5, -2.5, 0.25) to (5, 2.5, 0.25)
- Target 3: Moving along the path from (-5, -2.5, 0.25) to (-5, 2.5, 0.25)
- Goal Zone 1: Centered at (0, 7, 0.05)
- Goal Zone 2: Centered at (7, 0, 0.05)
- Goal Zone 3: Centered at (-7, 0, 0.05)

##### Success Conditions:

1458 The task is considered complete when all three targets come to rest fully  
 1459 within their respective goal zones.  
 1460

1461 **Time Limit:**  
 1462 The robot has a time limit of 5 minutes to complete the task.

1463 **Rewards:**  
 1464 – Provide a small reward for moving closer to a target.  
 1465 – Provide a moderate reward for reaching a target.  
 1466 – Provide a large reward for pushing a target towards its goal zone.  
 1467 – Provide a significant reward for delivering a target to its goal zone.  
 1468 – Apply a small penalty for collisions with the targets or the boundaries  
     of the area.

1469

1470 **Termination:**  
 1471 The episode terminates if the robot flips over, if a target is pushed out  
     of bounds, or if the time limit is exceeded. The episode should  
 1472 terminate with success if all targets are delivered to their goal  
 1473 zones.

## 1475 Task 176

1477 **Task:** Push two balls into separate goals while avoiding static obstacles.

1478 **Description:**

- 1479 – The environment consists of a large flat ground.
- 1480 – Two balls, one red and one blue, are placed at fixed initial positions  
     on the ground. The red ball is positioned at [2.0, -2.0, 0.5] and the  
     blue ball is positioned at [2.0, 2.0, 0.5].
- 1481 – Two goal areas are defined by pairs of posts. The red goal is defined  
     by posts placed at [10.0, -2.0, 1.0] and [10.0, -1.0, 1.0]. The blue  
     goal is defined by posts placed at [10.0, 1.0, 1.0] and [10.0, 2.0,  
     1.0].
- 1482 – Three cylindrical obstacles of 1 meter radius and 2 meters height are  
     placed at fixed positions: [5.0, 0.0, 1.0], [7.0, -1.0, 1.0], and  
     [7.0, 1.0, 1.0].
- 1483 – The robot is initialized at [0.0, 0.0, 0.5], facing the positive x–  
     direction.
- 1484 – The task is for the robot to push the red ball into the red goal and  
     the blue ball into the blue goal while avoiding the obstacles.

1485 **Success:**

1486 The task is completed successfully if the robot pushes both balls into  
 1487 their respective goals without colliding with any obstacles.

1488 **Rewards:**

- 1489 – The robot is rewarded for decreasing its distance to each ball.
- 1490 – The robot is rewarded for pushing each ball towards its respective goal  
     .
- 1491 – The robot is penalized for colliding with obstacles.
- 1492 – The robot is greatly rewarded if a ball passes between the goal posts  
     of its respective goal.

1493 **Termination:**

1494 The task is terminated if the robot collides with an obstacle, if a ball  
 1495 collides with an obstacle, or if a time limit of 60 seconds is  
 1496 reached.

## 1507 Task 178

1509 **Task:** Coordinate with another moving ball to hit a target

1510 **Description:**

- 1511 – The environment consists of a large flat ground.

1512     – Two balls are placed 5 meters in front of the robot, spaced 2 meters  
 1513        apart laterally. Both balls begin rolling in parallel at 1 m/s after  
 1514        a short delay of 1 second.  
 1515     – A target area defined by a 2x2 meter flat vertical target is placed 10  
 1516        meters from the balls' initial position. The target moves laterally  
 1517        back and forth at 0.5 m/s, covering 3 meters to each side before  
 1518        reversing direction.  
 1519     – The robot is initialized 3 meters behind the balls, facing the  
 1520        direction of the target.  
 1521     – The task is for the robot to coordinate its movements to kick both  
 1522        balls such that they hit the moving target simultaneously.

1523     **Success:**  
 1524     The task is completed successfully if both balls hit the front face of  
 1525        the moving target simultaneously.

1526     **Rewards:**  
 1527     – The robot is rewarded for decreasing the distance to each ball as they  
 1528        move.  
 1529     – The robot is rewarded for matching the velocity of each ball as it  
 1530        approaches.  
 1531     – The robot is rewarded for kicking each ball with a velocity that is  
 1532        likely to reach the target.  
 1533     – The robot is greatly rewarded if both balls hit the front face of the  
 1534        moving target simultaneously.

1535     **Termination:**  
 1536     The task is terminated if either ball stops moving, if the balls do not  
 1537        hit the target simultaneously, or if a time limit of 30 seconds is  
 1538        reached.

### Task 196

1539     **Task: Dodgeball Arena**

1540     **Description:**

- 1541     – The environment is a 15 m x 15 m walled arena with a flat ground  
       surface.
- 1542     – There are 5 automatic ball launchers positioned around the perimeter of  
       the arena. Each launcher can shoot a foam ball (0.2 m diameter) at  
       varying speeds (up to 5 m/s) and angles.
- 1543     – The launchers are programmed to shoot balls at random intervals (   
       between 1 to 3 seconds) and in random directions.
- 1544     – The robot starts in the center of the arena.

1545     **Task:**

1546     The robot must avoid being hit by the foam balls for as long as possible.  
 1547        The robot can move around the arena and jump to dodge the balls.

1548     **Success Conditions:**

1549     The task is considered successful if the robot can avoid being hit by any  
 1550        balls for a duration of 2 minutes.

1551     **Rewards:**

- 1552     – Provide a small reward for each second the robot avoids being hit.
- 1553     – Provide a large reward for successfully avoiding all balls for the  
       entire duration.
- 1554     – Apply a small penalty for each ball that hits the robot.

1555     **Termination:**

1556     The episode ends if the robot is hit by a ball, or if the time limit of 2  
 1557        minutes is exceeded.

### Task 197

1566

**Task: Jumping Over Gaps**

1567

**Description:**

1568

- The environment consists of a 10 m x 10 m platform with a series of gaps that the robot must jump over.
- The platform is divided into three sections:
  1. The first section (3 m x 10 m) has three gaps, each 0.5 m wide, spaced 2 m apart.
  2. The second section (3 m x 10 m) has two gaps, each 1 m wide, spaced 3 m apart.
  3. The third section (4 m x 10 m) has one gap, 1.5 m wide, located 2 m from the end of the platform.
- The robot begins at the start of the platform, facing the positive x-axis.

1569

**Task:**

1570

The robot must navigate through the three sections of the platform, jumping over the gaps to reach the end of the platform.

1571

**Success Conditions:**

1572

The task is considered complete when the robot reaches the end of the platform without falling into any gaps.

1573

**Rewards:**

1574

- Provide a small reward for each gap successfully jumped over.
- Provide a large reward for reaching the end of the platform.
- Apply a small penalty for each failed jump (falling into a gap).

1575

**Termination:**

1576

The episode ends if the robot falls into a gap, flips over, or reaches the end of the platform.

1577

1578

1579

1580

1581

1582

1583

1584

1585

1586

1587

1588

1589

1590

1591

1592

1593

1594

1595

1596

1597

1598

1599

1600

1601

1602

1603

1604

1605

1606

1607

1608

1609

1610

1611

1612

1613

1614

1615

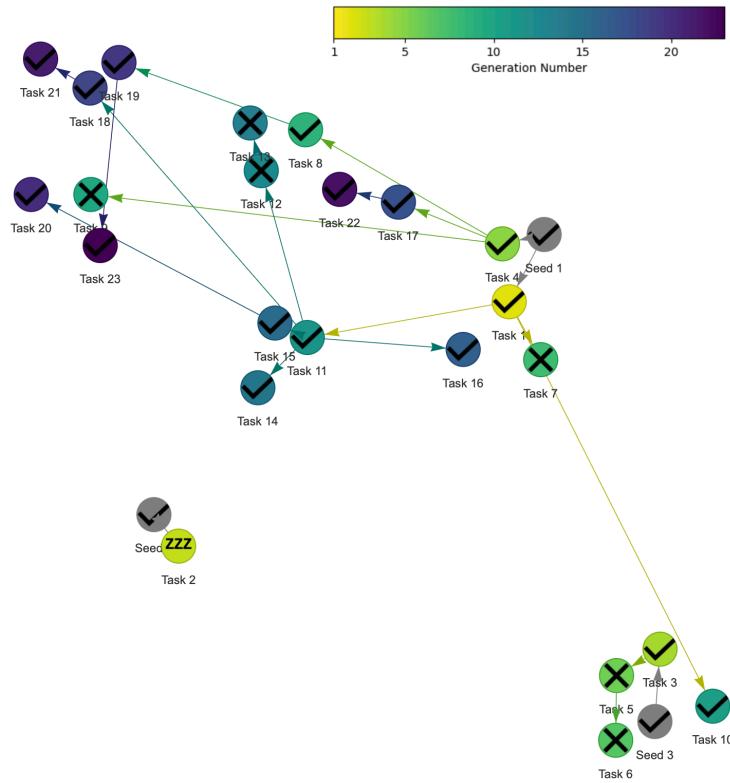
1616

1617

1618

1619

1620           **E SUPPLEMENTARY MATERIALS FOR SHORT RUN WITH LEARNING**  
 1621  
 1622



1649           **Figure 9: Short run with Learning task graph.** The node color reflects the generation number of  
 1650           the task. A node with a check mark indicates successful task learning, while a cross mark denotes  
 1651           a task that was attempted but failed to be learned. A ZZZ symbol means that the task was deemed  
 1652           uninteresting and discarded. The node connections illustrate which tasks were conditioned on when  
 1653           asking an FM to generate a similar yet new and interesting task.

1654           **Task 1**  
 1655

```
1656 import numpy as np
1657 from oped.envs.r2d2.base import R2D2Env
1658
1659 class Env(R2D2Env):
1660     """
1661         Cross a bridge with moving segments to reach a platform.
1662
1663     Description:
1664     - A start platform and an end platform (each 3 m in size and 0.5 m in
1665       thickness) are placed 20 m apart.
1666     - The two platforms are connected by a bridge (2 m wide) divided into
1667       5 segments of equal length (3 m each).
1668     - Each segment moves up and down independently with a sinusoidal
1669       motion. The amplitude is 1 m and the period is 2 seconds, with
1670       each segment offset in phase by 0.4 seconds from the previous one
1671
1672     - The segments are colored red, orange, yellow, green, and blue.
1673     The robot is initialized on the start platform.
1674     The task of the robot is to cross the dynamic bridge to reach the end
1675       platform as fast as possible by timing its jumps between the
1676       moving segments.
```

```

1674
1675
1676
1677
1678
1679
1680
1681
1682
1683
1684
1685
1686
1687
1688
1689
1690
1691
1692
1693
1694
1695
1696
1697
1698
1699
1700
1701
1702
1703
1704
1705
1706
1707
1708
1709
1710
1711
1712
1713
1714
1715
1716
1717
1718
1719
1720
1721
1722
1723
1724
1725
1726
1727

```

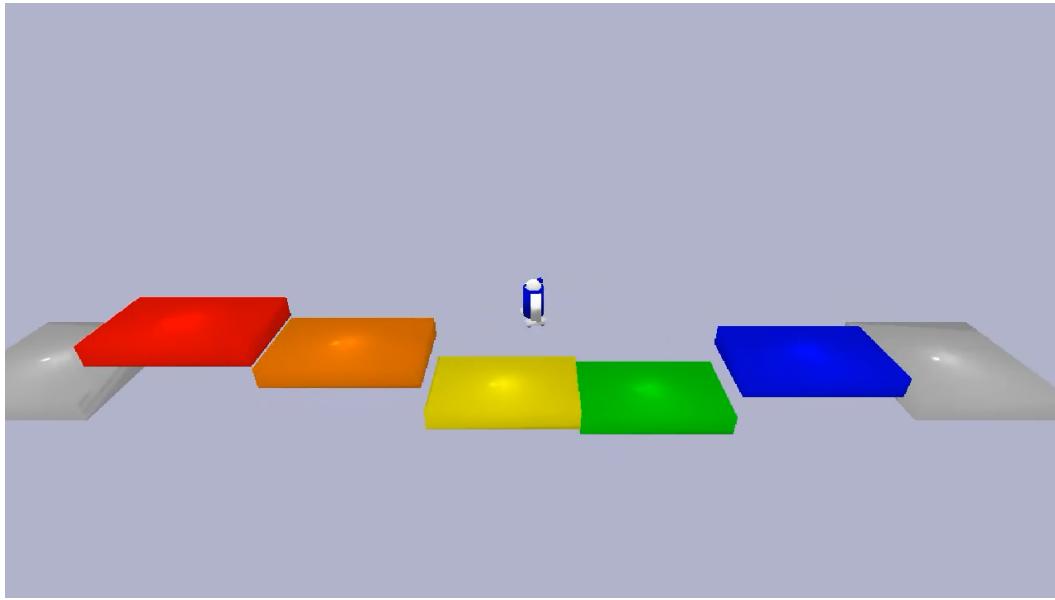


Figure 10: Task 1 of the OMNI-EPIC run presented in Section 5.

**Success:**  
The task is successfully completed when the robot reaches the end platform.

**Rewards:**  
To help the robot complete the task:

- The robot receives a reward for each time step it remains on the bridge or platforms, encouraging steady progress.
- The robot is rewarded based on how much it reduces the distance to the end platform, incentivizing swift movement towards the goal.

**Termination:**  
The task terminates immediately if the robot falls off the start platform, any segment of the bridge, or the end platform.  
\*\*\*\*

```

def __init__(self):
    super().__init__()
    self.platform_size = [3.0, 3.0, 0.5]
    self.platform_start_position = [0.0, 0.0, 0.0]
    self.platform_end_position = [self.platform_start_position[0] +
        20.0, self.platform_start_position[1], self.
        platform_start_position[2]]
    self.platform_start_id = self.create_box(mass=0.0, half_extents=[

        self.platform_size[0] / 2, self.platform_size[1] / 2, self.
        platform_size[2] / 2], position=self.platform_start_position,
        color=[0.8, 0.8, 0.8, 1.0])
    self.platform_end_id = self.create_box(mass=0.0, half_extents=[

        self.platform_size[0] / 2, self.platform_size[1] / 2, self.
        platform_size[2] / 2], position=self.platform_end_position,
        color=[0.8, 0.8, 0.8, 1.0])
    self.bridge_length = self.platform_end_position[0] - self.
        platform_start_position[0] - self.platform_size[0]
    self.bridge_width = 2.0
    self.num_segments = 5
    self.segment_length = self.bridge_length / self.num_segments
    self.segment_amplitude = 1.0
    self.segment_period = 2.0

```

```

1728     self.segment_phase_offset = 0.4
1729     segment_colors = [[1.0, 0.0, 0.0, 1.0], [1.0, 0.5, 0.0, 1.0],
1730                     [1.0, 1.0, 0.0, 1.0], [0.0, 1.0, 0.0, 1.0], [0.0, 0.0, 1.0,
1731                     1.0]]
1732     self.segment_ids = []
1733     for i in range(self.num_segments):
1734         segment_id = self.create_box(mass=0.0, half_extents=[self.
1735             segment_length / 2, self.bridge_width / 2, self.
1736             platform_size[2] / 2], position=[self.
1737             platform_start_position[0] + self.platform_size[0] / 2 +
1738             self.segment_length / 2 + i * self.segment_length, self.
1739             platform_start_position[1], self.platform_start_position
1740             [2]], color=segment_colors[i])
1741         self._p.changeDynamics(bodyUniqueId=segment_id, linkIndex=-1,
1742             lateralFriction=0.8, restitution=0.5)
1743         self.segment_ids.append(segment_id)
1744
1745     def create_box(self, mass, half_extents, position, color):
1746         collision_shape_id = self._p.createCollisionShape(shapeType=self._p.GEOM_BOX, halfExtents=half_extents)
1747         visual_shape_id = self._p.createVisualShape(shapeType=self._p.GEOM_BOX, halfExtents=half_extents, rgbaColor=color)
1748         return self._p.createMultiBody(baseMass=mass,
1749             baseCollisionShapeIndex=collision_shape_id,
1750             baseVisualShapeIndex=visual_shape_id, basePosition=position)
1751
1752     def get_object_position(self, object_id):
1753         return np.asarray(self._p.getBasePositionAndOrientation(object_id)[0])
1754
1755     def get_distance_to_object(self, object_id):
1756         object_position = self.get_object_position(object_id)
1757         robot_position = self.robot.links['base'].position
1758         return np.linalg.norm(object_position[:2] - robot_position[:2])
1759
1760     def reset(self):
1761         observation = super().reset()
1762         self.time = 0.0
1763         self._p.resetBasePositionAndOrientation(self.robot.robot_id, [
1764             self.platform_start_position[0], self.platform_start_position[1],
1765             self.platform_start_position[2] + self.platform_size[2] / 2 +
1766             self.robot.links['base'].position_init[2]], self.robot.
1767             links['base'].orientation_init)
1768         return observation
1769
1770     def step(self, action):
1771         self.distance_to_platform_end = self.get_distance_to_object(self.
1772             platform_end_id)
1773         observation, reward, terminated, truncated, info = super().step(
1774             action)
1775         self.time += self.dt
1776         for i, segment_id in enumerate(self.segment_ids):
1777             segment_position = self.get_object_position(segment_id)
1778             new_segment_position = [segment_position[0], segment_position[1],
1779             self.platform_start_position[2] + self.
1780             segment_amplitude * np.sin(2 * np.pi * (self.time + i *
1781             self.segment_phase_offset) / self.segment_period)]
1782             self._p.resetBasePositionAndOrientation(segment_id,
1783             new_segment_position, [0.0, 0.0, 0.0, 1.0])
1784         return (observation, reward, terminated, truncated, info)
1785
1786     def get_task_rewards(self, action):
1787         new_distance_to_platform_end = self.get_distance_to_object(self.
1788             platform_end_id)

```

```

1782     on_bridge_or_platforms = 1.0 if self.robot.links['base'].position
1783         [2] > self.platform_start_position[2] + self.platform_size[2]
1784             / 2 else -1.0
1785     reach_platform_end = (self.distance_to_platform_end -
1786         new_distance_to_platform_end) / self.dt
1787     return {'on_bridge_or_platforms': on_bridge_or_platforms, '
1788             'reach_platform_end': reach_platform_end}
1789
1790     def get_terminated(self, action):
1791         return self.robot.links['base'].position[2] < self.
1792             platform_start_position[2]
1793
1794     def get_success(self):
1795         is_on_platform_end = self.get_distance_to_object(self.
1796             platform_end_id) < self.platform_size[2] / 2
1797         return is_on_platform_end

```

### Task 3

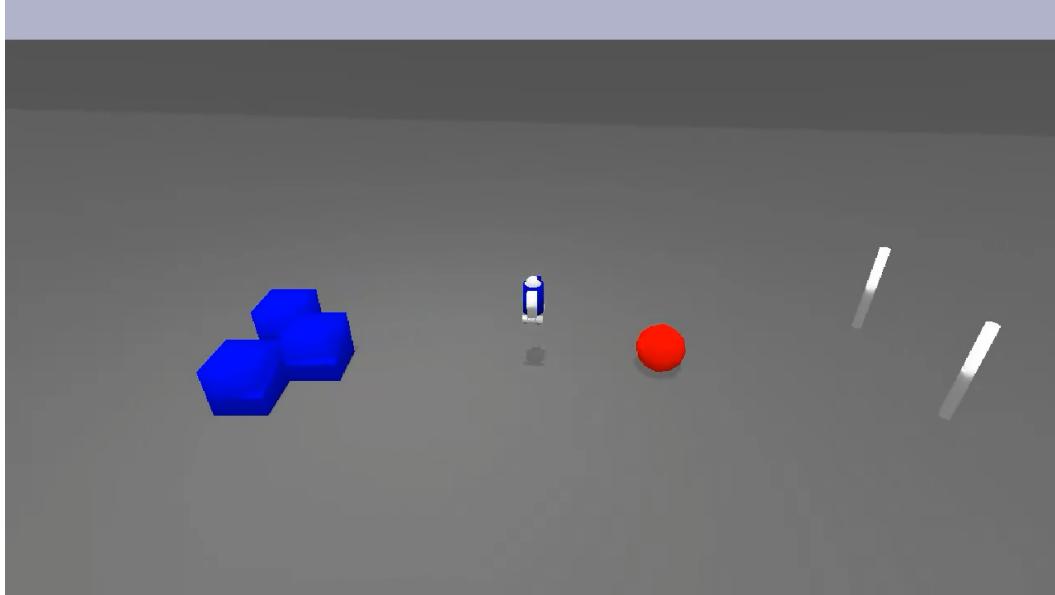


Figure 11: Task 3 of the OMNI-EPIC run presented in Section 5.

```

1818
1819
1820
1821 import numpy as np
1822 from oped.envs.r2d2.base import R2D2Env
1823
1824 class Env(R2D2Env):
1825     """
1826     Kick a ball into a goal with obstacles.
1827
1828     Description:
1829     - The environment consists of a large flat ground measuring 1000 x
1830       1000 x 10 meters.
1831     - A ball with a radius of 0.5 meters is placed randomly on the ground
1832       , 5 meters in front of the robot's starting position.
1833     - The goal is defined by two goal posts, each 2 meters high and
1834       placed 3 meters apart, forming a goal area. The goal is
1835       positioned 10 meters in front of the robot's starting position.
1836     - Three box-shaped obstacles, each measuring 1 x 1 x 1 meters, are
1837       placed between the robot's starting position and the ball.
1838     - The first obstacle is placed 2 meters in front and 1 meter to the
1839       left of the robot.

```

```

1836
1837     - The second obstacle is placed 3 meters in front of the robot.
1838     - The third obstacle is placed 2 meters in front and 1 meter to the
1839         right of the robot.
1840     - The robot is initialized at a fixed position on the ground, facing
1841         the goal.
1842     - The task of the robot is to navigate around the obstacles, reach
1843         the ball, and kick it into the goal.
1844
1845 Success:
1846 The task is successfully completed if the robot kicks the ball so
1847     that it passes between the two goal posts without colliding with
1848     any of the obstacles.
1849
1850 Rewards:
1851 To help the robot complete the task:
1852     - The robot is rewarded for survival at each time step.
1853     - The robot is rewarded for decreasing its distance to the ball while
1854         avoiding obstacles.
1855     - The robot is penalized for colliding with obstacles.
1856     - The robot is rewarded for kicking the ball towards the goal, with
1857         additional rewards for successfully kicking the ball into the
1858         goal.
1859
1860 Termination:
1861 The task is terminated if the robot collides with any obstacle.
1862     Otherwise, it does not have a specific termination condition.
1863 """
1864
1865 def __init__(self):
1866     super().__init__()
1867     self.ground_size = [1000.0, 1000.0, 10.0]
1868     self.ground_position = [0.0, 0.0, 0.0]
1869     self.ground_id = self.create_box(mass=0.0, half_extents=[self.
1870         ground_size[0] / 2, self.ground_size[1] / 2, self.ground_size
1871         [2] / 2], position=self.ground_position, color=[0.5, 0.5,
1872         0.5, 1.0])
1873     self._p.changeDynamics(bodyUniqueId=self.ground_id, linkIndex=-1,
1874         lateralFriction=0.8, restitution=0.5)
1875     self.ball_radius = 0.5
1876     self.ball_id = self.create_sphere(mass=1.0, radius=self.
1877         ball_radius, position=[0.0, 0.0, 0.0], color=[1.0, 0.0, 0.0,
1878         1.0])
1879     self.goal_post_height = 2.0
1880     self.goal_post_radius = 0.1
1881     self.goal_width = 3.0
1882     self.goal_position = [10.0, 0.0, self.ground_position[2] + self.
1883         ground_size[2] / 2 + self.goal_post_height / 2]
1884     self.goal_post_left_id = self.create_cylinder(mass=0.0, radius=
1885         self.goal_post_radius, height=self.goal_post_height, position
1886         =[self.goal_position[0], self.goal_position[1] - self.
1887             goal_width / 2, self.goal_position[2]], color=[1.0, 1.0, 1.0,
1888             1.0])
1889     self.goal_post_right_id = self.create_cylinder(mass=0.0, radius=
1890         self.goal_post_radius, height=self.goal_post_height, position
1891         =[self.goal_position[0], self.goal_position[1] + self.
1892             goal_width / 2, self.goal_position[2]], color=[1.0, 1.0, 1.0,
1893             1.0])
1894     self.obstacle_size = [1.0, 1.0, 1.0]
1895     self.obstacle_1_id = self.create_box(mass=0.0, half_extents=[self.
1896         obstacle_size[0] / 2, self.obstacle_size[1] / 2, self.
1897         ground_position[2] + self.ground_size[2] / 2 + self.
1898         obstacle_size[2] / 2], position=[-3.0, -1.0, self.
1899             ground_size[2] / 2], color=[0.0, 0.0, 1.0, 1.0])
1900     self.obstacle_2_id = self.create_box(mass=0.0, half_extents=[self.
1901         obstacle_size[0] / 2, self.obstacle_size[1] / 2, self.
1902             ground_size[2] / 2], position=[-1.0, 1.0, self.
1903                 ground_size[2] / 2], color=[0.0, 1.0, 1.0, 1.0])

```

```

1890     obstacle_size[2] / 2], position=[-2.0, 0.0, self.
1891     ground_position[2] + self.ground_size[2] / 2 + self.
1892     obstacle_size[2] / 2], color=[0.0, 0.0, 1.0, 1.0])
1893     self.obstacle_3_id = self.create_box(mass=0.0, half_extents=[self.
1894     .obstacle_size[0] / 2, self.obstacle_size[1] / 2, self.
1895     obstacle_size[2] / 2], position=[-3.0, 1.0, self.
1896     ground_position[2] + self.ground_size[2] / 2 + self.
1897     obstacle_size[2] / 2], color=[0.0, 0.0, 1.0, 1.0])
1898     self.robot_position_init = [0.0, 0.0, self.ground_position[2] +
1899     self.ground_size[2] / 2 + self.robot.links['base']].
1900     position_init[2]
1901     self.robot_orientation_init = self._p.getQuaternionFromEuler
1902     ([0.0, 0.0, 0.0])
1903
1904     def create_box(self, mass, half_extents, position, color):
1905         collision_shape_id = self._p.createCollisionShape(shapeType=self.
1906             ._p.GEOM_BOX, halfExtents=half_extents)
1907         visual_shape_id = self._p.createVisualShape(shapeType=self._p.
1908             GEOM_BOX, halfExtents=half_extents, rgbaColor=color)
1909         return self._p.createMultiBody(baseMass=mass,
1910             baseCollisionShapeIndex=collision_shape_id,
1911             baseVisualShapeIndex=visual_shape_id, basePosition=position)
1912
1913     def create_sphere(self, mass, radius, position, color):
1914         collision_shape_id = self._p.createCollisionShape(shapeType=self.
1915             ._p.GEOM_SPHERE, radius=radius)
1916         visual_shape_id = self._p.createVisualShape(shapeType=self._p.
1917             GEOM_SPHERE, radius=radius, rgbaColor=color)
1918         return self._p.createMultiBody(baseMass=mass,
1919             baseCollisionShapeIndex=collision_shape_id,
1920             baseVisualShapeIndex=visual_shape_id, basePosition=position)
1921
1922     def create_cylinder(self, mass, radius, height, position, color):
1923         collision_shape_id = self._p.createCollisionShape(shapeType=self.
1924             ._p.GEOM_CYLINDER, radius=radius, height=height)
1925         visual_shape_id = self._p.createVisualShape(shapeType=self._p.
1926             GEOM_CYLINDER, radius=radius, length=height, rgbaColor=color)
1927         return self._p.createMultiBody(baseMass=mass,
1928             baseCollisionShapeIndex=collision_shape_id,
1929             baseVisualShapeIndex=visual_shape_id, basePosition=position)
1930
1931     def get_object_position(self, object_id):
1932         return np.asarray(self._p.getBasePositionAndOrientation(object_id
1933             )[0])
1934
1935     def get_distance_to_object(self, object_id):
1936         object_position = self.get_object_position(object_id)
1937         robot_position = self.robot.links['base'].position
1938         return np.linalg.norm(object_position[:2] - robot_position[:2])
1939
1940     def reset(self):
1941         observation = super().reset()
1942         self._p.resetBasePositionAndOrientation(self.robot.robot_id, self.
1943             .robot_position_init, self.robot_orientation_init)
1944         ball_y_init = np.random.uniform(-2.0, 2.0)
1945         self._p.resetBasePositionAndOrientation(self.ball_id, [5.0,
1946             ball_y_init, self.ground_position[2] + self.ground_size[2] /
1947             2 + self.ball_radius], [0.0, 0.0, 0.0, 1.0])
1948         return observation
1949
1950     def step(self, action):
1951         self.distance_to_ball = self.get_distance_to_object(self.ball_id)
1952         self.ball_position = self.get_object_position(self.ball_id)
1953         observation, reward, terminated, truncated, info = super().step(
1954             action)

```

```

1944     return (observation, reward, terminated, truncated, info)
1945
1946     def get_task_rewards(self, action):
1947         new_distance_to_ball = self.get_distance_to_object(self.ball_id)
1948         new_ball_position = self.get_object_position(self.ball_id)
1949         survival = 1.0
1950         reach_ball = (self.distance_to_ball - new_distance_to_ball) /
1951             self.dt
1952         collision_obstacle_1 = len(self._p.getContactPoints(bodyA=self.
1953             robot.robot_id, bodyB=self.obstacle_1_id)) > 0
1954         collision_obstacle_2 = len(self._p.getContactPoints(bodyA=self.
1955             robot.robot_id, bodyB=self.obstacle_2_id)) > 0
1956         collision_obstacle_3 = len(self._p.getContactPoints(bodyA=self.
1957             robot.robot_id, bodyB=self.obstacle_3_id)) > 0
1958         collision_penalty = -10.0 if collision_obstacle_1 or
1959             collision_obstacle_2 or collision_obstacle_3 else 0.0
1960         kick_ball = (new_ball_position[0] - self.ball_position[0]) / self
1961             .dt
1962         ball_in_goal = 0.0
1963         if self.goal_position[1] - self.goal_width / 2 <
1964             new_ball_position[1] < self.goal_position[1] + self.
1965                 goal_width / 2 and new_ball_position[0] > self.goal_position
1966                 [0]:
1967                     ball_in_goal = 10.0
1968         return {'survival': survival, 'reach_ball': reach_ball, '
1969             'collision_penalty': collision_penalty, 'kick_ball': kick_ball
1970             , 'ball_in_goal': ball_in_goal}
1971
1972     def get_terminated(self, action):
1973         collision_obstacle_1 = len(self._p.getContactPoints(bodyA=self.
1974             robot.robot_id, bodyB=self.obstacle_1_id)) > 0
1975         collision_obstacle_2 = len(self._p.getContactPoints(bodyA=self.
1976             robot.robot_id, bodyB=self.obstacle_2_id)) > 0
1977         collision_obstacle_3 = len(self._p.getContactPoints(bodyA=self.
1978             robot.robot_id, bodyB=self.obstacle_3_id)) > 0
1979         return collision_obstacle_1 or collision_obstacle_2 or
1980             collision_obstacle_3
1981
1982     def get_success(self):
1983         ball_position = self.get_object_position(self.ball_id)
1984         return self.goal_position[1] - self.goal_width / 2 <
1985             ball_position[1] < self.goal_position[1] + self.goal_width /
1986                 2 and ball_position[0] > self.goal_position[0]

```

#### Task 4

```

1983 import numpy as np
1984 from oped.envs.r2d2.base import R2D2Env
1985
1986 class Env(R2D2Env):
1987     """
1988         Cross a rainbow bridge with moving segments and gaps to reach a
1989         platform.
1990
1991         Description:
1992             - A start platform and an end platform (each 3 m in size and 0.5 m in
1993                 thickness) are placed 25 m apart.
1994             - The two platforms are connected by a bridge (2 m wide) divided into
1995                 6 segments of equal length (3 m each).
1996             - Each segment moves up and down independently with a sinusoidal
1997                 motion. The amplitude is 1 m and the period is 2 seconds, with
1998                 each segment offset in phase by 1/3 seconds from the previous one
1999                 .
2000             - The segments are colored red, orange, yellow, green, blue, and
2001                 purple, like a rainbow.

```

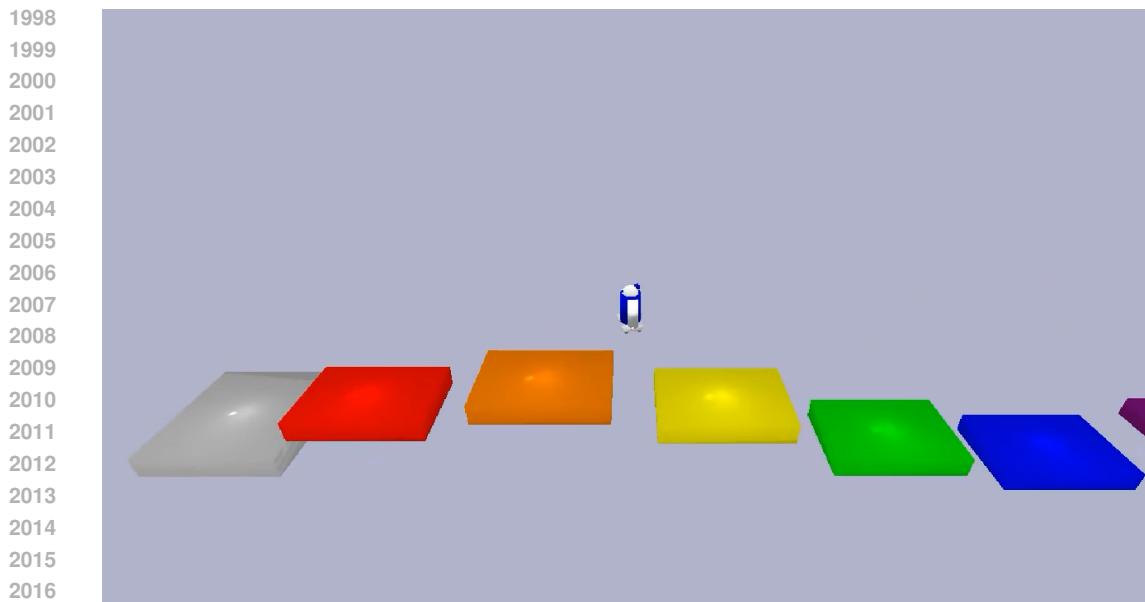


Figure 12: Task 4 of the OMNI-EPIC run presented in Section 5.

1998  
 1999  
 2000  
 2001  
 2002  
 2003  
 2004  
 2005  
 2006  
 2007  
 2008  
 2009  
 2010  
 2011  
 2012  
 2013  
 2014  
 2015  
 2016  
 2017  
 2018  
 2019  
 2020  
 2021  
 2022     – The segments have 1 m gaps between them.  
 The robot is initialized on the start platform.  
 2023     The task of the robot is to cross the dynamic bridge to reach the end  
 2024       platform as fast as possible by timing its jumps between the  
 2025       moving segments and over the gaps.  
 2026  
 2027     Success:  
 2028       The task is successfully completed when the robot reaches the end  
 2029       platform.  
 2030  
 2031     Rewards:  
 2032       To help the robot complete the task:  
 2033       – The robot receives a reward for each time step it remains on the  
 2034       bridge or platforms, encouraging steady progress.  
 2035       – The robot is rewarded based on how much it reduces the distance to  
 2036       the end platform, incentivizing swift movement towards the goal.  
 2037  
 2038     Termination:  
 2039       The task terminates immediately if the robot falls off the start  
 2040       platform, any segment of the bridge, or the end platform.  
 2041  
 2042     def \_\_init\_\_(self):  
 2043       super().\_\_init\_\_()  
 2044       self.platform\_size = [3.0, 3.0, 0.5]  
 2045       self.platform\_start\_position = [0.0, 0.0, 0.0]  
 2046       self.platform\_end\_position = [self.platform\_start\_position[0] +  
 2047           25.0, self.platform\_start\_position[1], self.  
 2048           platform\_start\_position[2]]  
 2049       self.platform\_start\_id = self.create\_box(mass=0.0, half\_extents=[  
 2050           self.platform\_size[0] / 2, self.platform\_size[1] / 2, self.  
 2051           platform\_size[2] / 2], position=self.platform\_start\_position,  
 2052           color=[0.8, 0.8, 0.8, 1.0])  
 2053       self.platform\_end\_id = self.create\_box(mass=0.0, half\_extents=[  
 2054           self.platform\_size[0] / 2, self.platform\_size[1] / 2, self.  
 2055           platform\_size[2] / 2], position=self.platform\_end\_position,  
 2056           color=[0.8, 0.8, 0.8, 1.0])

```

2052         self.bridge_length = self.platform_end_position[0] - self.
2053             platform_start_position[0] - self.platform_size[0]
2054         self.bridge_width = 2.0
2055         self.num_segments = 6
2056         self.segment_length = 3.0
2057         self.gap_length = 1.0
2058         self.segment_amplitude = 1.0
2059         self.segment_period = 2.0
2060         self.segment_phase_offset = 1.0 / 3.0
2061         segment_colors = [[1.0, 0.0, 0.0, 1.0], [1.0, 0.5, 0.0, 1.0],
2062             [1.0, 1.0, 0.0, 1.0], [0.0, 1.0, 0.0, 1.0], [0.0, 0.0, 1.0,
2063                 1.0], [0.5, 0.0, 0.5, 1.0]]
2064         self.segment_ids = []
2065         for i in range(self.num_segments):
2066             segment_id = self.create_box(mass=0.0, half_extents=[self.
2067                 segment_length / 2, self.bridge_width / 2, self.
2068                 platform_size[2] / 2], position=[self.
2069                     platform_start_position[0] + self.platform_size[0] / 2 +
2070                         self.segment_length / 2 + i * (self.segment_length + self.
2071                             .gap_length), self.platform_start_position[1], self.
2072                             platform_start_position[2]], color=segment_colors[i])
2073             self._p.changeDynamics(bodyUniqueId=segment_id, linkIndex=-1,
2074                 lateralFriction=0.8, restitution=0.5)
2075             self.segment_ids.append(segment_id)
2076
2077     def create_box(self, mass, half_extents, position, color):
2078         collision_shape_id = self._p.createCollisionShape(shapeType=self._p.GEOM_BOX, halfExtents=half_extents)
2079         visual_shape_id = self._p.createVisualShape(shapeType=self._p.GEOM_BOX, halfExtents=half_extents, rgbaColor=color)
2080         return self._p.createMultiBody(baseMass=mass,
2081             baseCollisionShapeIndex=collision_shape_id,
2082             baseVisualShapeIndex=visual_shape_id, basePosition=position)
2083
2084     def get_object_position(self, object_id):
2085         return np.asarray(self._p.getBasePositionAndOrientation(object_id)[0])
2086
2087     def get_distance_to_object(self, object_id):
2088         object_position = self.get_object_position(object_id)
2089         robot_position = self.robot.links['base'].position
2090         return np.linalg.norm(object_position[:2] - robot_position[:2])
2091
2092     def reset(self):
2093         observation = super().reset()
2094         self.time = 0.0
2095         self._p.resetBasePositionAndOrientation(self.robot.robot_id, [
2096             self.platform_start_position[0], self.platform_start_position[1],
2097             self.platform_start_position[2] + self.platform_size[2] / 2 +
2098                 self.robot.links['base'].position_init[2]], self.robot.links['base'].orientation_init)
2099         return observation
2100
2101     def step(self, action):
2102         self.distance_to_platform_end = self.get_distance_to_object(self.
2103             platform_end_id)
2104         observation, reward, terminated, truncated, info = super().step(
2105             action)
2106         self.time += self.dt
2107         for i, segment_id in enumerate(self.segment_ids):
2108             segment_position = self.get_object_position(segment_id)
2109             new_segment_position = [segment_position[0], segment_position[1],
2110                 self.platform_start_position[2] + self.
2111                     segment_amplitude * np.sin(2 * np.pi * (self.time + i *
2112                         self.segment_phase_offset) / self.segment_period)]

```

```

2106         self._p.resetBasePositionAndOrientation(segment_id,
2107             new_segment_position, [0.0, 0.0, 0.0, 1.0])
2108     return (observation, reward, terminated, truncated, info)
2109
2110     def get_task_rewards(self, action):
2111         new_distance_to_platform_end = self.get_distance_to_object(self.
2112             platform_end_id)
2113         on_bridge_or_platforms = 1.0 if self.robot.links['base'].position
2114             [2] > self.platform_start_position[2] + self.platform_size[2]
2115             / 2 else -1.0
2116         reach_platform_end = (self.distance_to_platform_end -
2117             new_distance_to_platform_end) / self.dt
2118         return {'on_bridge_or_platforms': on_bridge_or_platforms, '
2119             'reach_platform_end': reach_platform_end}
2120
2121     def get_terminated(self, action):
2122         return self.robot.links['base'].position[2] < self.
2123             platform_start_position[2]
2124
2125     def get_success(self):
2126         is_on_platform_end = self.get_distance_to_object(self.
2127             platform_end_id) < self.platform_size[2] / 2
2128         return is_on_platform_end

```

### Task 5

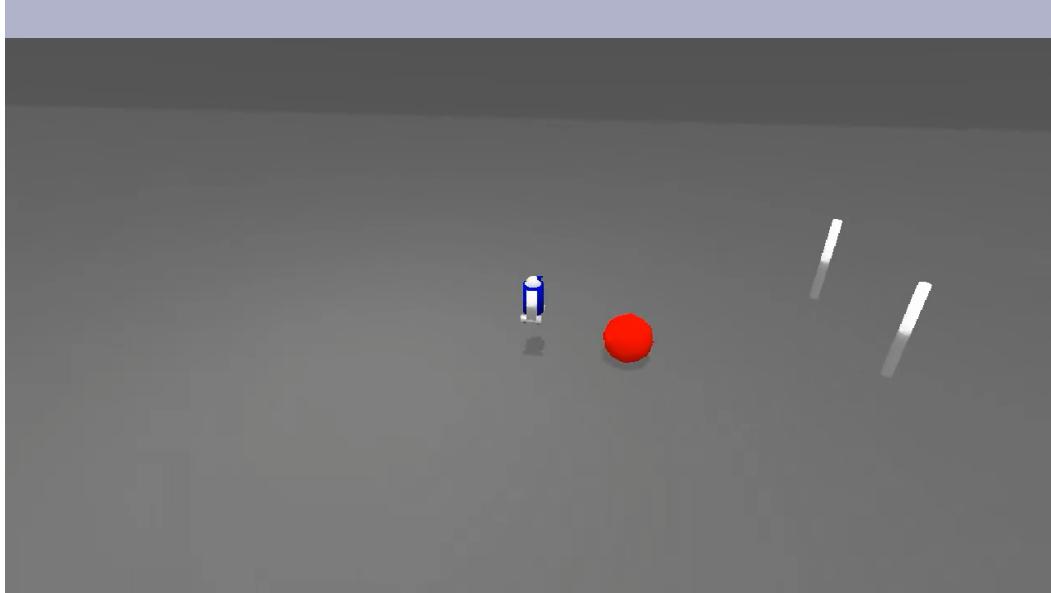


Figure 13: Task 5 of the OMNI-EPIC run presented in Section 5.

```

2149
2150     import numpy as np
2151     from oped.envs.r2d2.base import R2D2Env
2152
2153     class Env(R2D2Env):
2154         """
2155             Kick a ball into a moving goal.
2156
2157             Description:
2158             - The environment consists of a large flat ground measuring 1000 x
2159                 1000 x 10 meters.
2160             - A ball with a radius of 0.5 meters is placed on the ground, 5
2161                 meters directly in front of the robot's starting position.

```

```

2160
2161     - The goal is defined by two goal posts, each 2 meters high and
2162       placed 3 meters apart, forming a goal area.
2163     - The goal moves side-to-side (along the y-axis) in a sinusoidal
2164       motion with an amplitude of 2 meters and a period of 10 seconds.
2165       The center of its motion is 10 meters in front of the robot's
2166       starting position.
2167     - The robot is initialized at a fixed position on the ground, facing
2168       the goal.
2169     - The task of the robot is to move towards the ball, time its kick
2170       correctly, and kick the ball into the moving goal.
2171
2172 Success:
2173   The task is successfully completed if the robot kicks the ball so
2174     that it passes between the two goal posts.
2175
2176 Rewards:
2177   To help the robot complete the task:
2178     - The robot is rewarded for survival at each time step.
2179     - The robot is rewarded for decreasing its distance to the ball.
2180     - The robot is rewarded for kicking the ball towards the goal, with
2181       additional rewards for successfully kicking the ball into the
2182       goal.
2183
2184 Termination:
2185   The task does not have a specific termination condition. The episode
2186     ends after a fixed amount of time.
2187 """
2188
2189 def __init__(self):
2190     super().__init__()
2191     self.ground_size = [1000.0, 1000.0, 10.0]
2192     self.ground_position = [0.0, 0.0, 0.0]
2193     self.ground_id = self.create_box(mass=0.0, half_extents=[self.
2194       ground_size[0] / 2, self.ground_size[1] / 2, self.ground_size
2195       [2] / 2], position=self.ground_position, color=[0.5, 0.5,
2196       0.5, 1.0])
2197     self._p.changeDynamics(bodyUniqueId=self.ground_id, linkIndex=-1,
2198       lateralFriction=0.8, restitution=0.5)
2199     self.ball_radius = 0.5
2200     self.ball_id = self.create_sphere(mass=1.0, radius=self.
2201       ball_radius, position=[0.0, 0.0, 0.0], color=[1.0, 0.0, 0.0,
2202       1.0])
2203     self.goal_post_height = 2.0
2204     self.goal_post_radius = 0.1
2205     self.goal_width = 3.0
2206     self.goal_position_init = [10.0, 0.0, self.ground_position[2] +
2207       self.ground_size[2] / 2 + self.goal_post_height / 2]
2208     self.goal_post_left_id = self.create_cylinder(mass=0.0, radius=
2209       self.goal_post_radius, height=self.goal_post_height, position
2210       =[self.goal_position_init[0], self.goal_position_init[1] -
2211         self.goal_width / 2, self.goal_position_init[2]], color=[1.0,
2212       1.0, 1.0, 1.0])
2213     self.goal_post_right_id = self.create_cylinder(mass=0.0, radius=
2214       self.goal_post_radius, height=self.goal_post_height, position
2215       =[self.goal_position_init[0], self.goal_position_init[1] +
2216         self.goal_width / 2, self.goal_position_init[2]], color=[1.0,
2217       1.0, 1.0, 1.0])
2218     self.goal_amplitude = 2.0
2219     self.goal_period = 10.0
2220     self.robot_position_init = [self.ground_position[0], self.
2221       ground_position[1], self.ground_position[2] + self.
2222       ground_size[2] / 2 + self.robot.links['base'].position_init
2223       [2]]
2224
2225 def create_box(self, mass, half_extents, position, color):

```

```

2214     collision_shape_id = self._p.createCollisionShape(shapeType=self.
2215         _p.GEOM_BOX, halfExtents=half_extents)
2216     visual_shape_id = self._p.createVisualShape(shapeType=self._p.
2217         GEOM_BOX, halfExtents=half_extents, rgbaColor=color)
2218     return self._p.createMultiBody(baseMass=mass,
2219         baseCollisionShapeIndex=collision_shape_id,
2220         baseVisualShapeIndex=visual_shape_id, basePosition=position)
2221
2222     def create_sphere(self, mass, radius, position, color):
2223         collision_shape_id = self._p.createCollisionShape(shapeType=self.
2224             _p.GEOM_SPHERE, radius=radius)
2225         visual_shape_id = self._p.createVisualShape(shapeType=self._p.
2226             GEOM_SPHERE, radius=radius, rgbaColor=color)
2227         return self._p.createMultiBody(baseMass=mass,
2228             baseCollisionShapeIndex=collision_shape_id,
2229             baseVisualShapeIndex=visual_shape_id, basePosition=position)
2230
2231     def create_cylinder(self, mass, radius, height, position, color):
2232         collision_shape_id = self._p.createCollisionShape(shapeType=self.
2233             _p.GEOM_CYLINDER, radius=radius, height=height)
2234         visual_shape_id = self._p.createVisualShape(shapeType=self._p.
2235             GEOM_CYLINDER, radius=radius, length=height, rgbaColor=color)
2236         return self._p.createMultiBody(baseMass=mass,
2237             baseCollisionShapeIndex=collision_shape_id,
2238             baseVisualShapeIndex=visual_shape_id, basePosition=position)
2239
2240     def get_object_position(self, object_id):
2241         return np.asarray(self._p.getBasePositionAndOrientation(object_id
2242             )[0])
2243
2244     def get_distance_to_object(self, object_id):
2245         object_position = self.get_object_position(object_id)
2246         robot_position = self.robot.links['base'].position
2247         return np.linalg.norm(object_position[:2] - robot_position[:2])
2248
2249     def reset(self):
2250         observation = super().reset()
2251         self.time = 0.0
2252         self._p.resetBasePositionAndOrientation(self.robot.robot_id, self.
2253             .robot_position_init, self.robot.links['base'].
2254             orientation_init)
2255         self._p.resetBasePositionAndOrientation(self.ball_id, [self.
2256             .robot_position_init[0] + 5.0, self.robot_position_init[1],
2257             self.ground_position[2] + self.ground_size[2] / 2 + self.
2258             ball_radius], [0.0, 0.0, 0.0, 1.0])
2259         return observation
2260
2261     def step(self, action):
2262         self.distance_to_ball = self.get_distance_to_object(self.ball_id)
2263         self.ball_position = self.get_object_position(self.ball_id)
2264         observation, reward, terminated, truncated, info = super().step(
2265             action)
2266         self.time += self.dt
2267         goal_y = self.goal_amplitude * np.sin(2 * np.pi * self.time /
2268             self.goal_period)
2269         self._p.resetBasePositionAndOrientation(self.goal_post_left_id, [
2270             self.goal_position_init[0], self.goal_position_init[1] - self.
2271             .goal_width / 2 + goal_y, self.goal_position_init[2]], [0.0,
2272             0.0, 0.0, 1.0])
2273         self._p.resetBasePositionAndOrientation(self.goal_post_right_id,
2274             [self.goal_position_init[0], self.goal_position_init[1] +
2275                 self.goal_width / 2 + goal_y, self.goal_position_init[2]],
2276             [0.0, 0.0, 0.0, 1.0])
2277         return (observation, reward, terminated, truncated, info)

```

```

2268     def get_task_rewards(self, action):
2269         new_distance_to_ball = self.get_distance_to_object(self.ball_id)
2270         new_ball_position = self.get_object_position(self.ball_id)
2271         survival = 1.0
2272         reach_ball = (self.distance_to_ball - new_distance_to_ball) /
2273             self.dt
2274         kick_ball = (new_ball_position[0] - self.ball_position[0]) / self.
2275             .dt
2276         goal_y = self.goal_amplitude * np.sin(2 * np.pi * self.time /
2277             self.goal_period)
2278         ball_in_goal = 0.0
2279         if self.goal_position_init[0] - self.goal_post_radius <
2280             new_ball_position[0] < self.goal_position_init[0] + self.
2281             goal_post_radius and self.goal_position_init[1] - self.
2282             goal_width / 2 + goal_y < new_ball_position[1] < self.
2283             goal_position_init[1] + self.goal_width / 2 + goal_y:
2284             ball_in_goal = 10.0
2285         return {'survival': survival, 'reach_ball': reach_ball, '
2286             'kick_ball': kick_ball, 'ball_in_goal': ball_in_goal}
2287
2288     def get_terminated(self, action):
2289         return False
2290
2291     def get_success(self):
2292         ball_position = self.get_object_position(self.ball_id)
2293         goal_y = self.goal_amplitude * np.sin(2 * np.pi * self.time /
2294             self.goal_period)
2295         return self.goal_position_init[0] - self.goal_post_radius <
2296             ball_position[0] < self.goal_position_init[0] + self.
2297             goal_post_radius and self.goal_position_init[1] - self.
2298             goal_width / 2 + goal_y < ball_position[1] < self.
2299             goal_position_init[1] + self.goal_width / 2 + goal_y

```

### Task 6

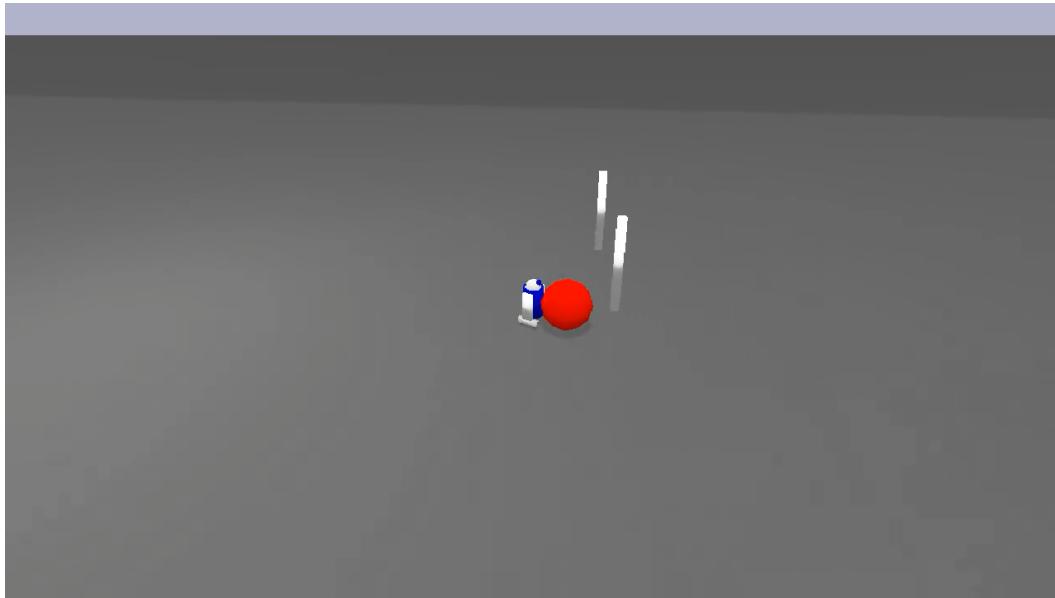


Figure 14: Task 6 of the OMNI-EPIC run presented in Section 5.

```

2318     import numpy as np
2319     from oped.envs.r2d2.base import R2D2Env

```

```

2322 class Env(R2D2Env):
2323     """
2324     Kick a ball into a moving goal.
2325
2326     Description:
2327     - The environment consists of a large flat ground measuring 1000 x
2328         1000 x 10 meters.
2329     - A ball with a radius of 0.5 meters is placed on the ground, 5
2330         meters directly in front of the robot's starting position.
2331     - The goal is defined by two goal posts, each 2 meters high and
2332         placed 3 meters apart, forming a goal area.
2333     - The goal moves side-to-side (along the y-axis) in a sinusoidal
2334         motion with an amplitude of 2 meters and a period of 10 seconds.
2335         The center of its motion is 10 meters in front of the robot's
2336         starting position.
2337     - The robot is initialized at a fixed position on the ground, facing
2338         the goal.
2339     - The task of the robot is to move towards the ball, time its kick
2340         correctly, and kick the ball into the moving goal.
2341
2342     Success:
2343     The task is successfully completed if the robot kicks the ball so
2344         that it passes between the two goal posts.
2345
2346     Rewards:
2347     To help the robot complete the task:
2348     - The robot is rewarded for survival at each time step.
2349     - The robot is rewarded for decreasing its distance to the ball.
2350     - The robot is rewarded for kicking the ball towards the goal, with
2351         additional rewards for successfully kicking the ball into the
2352         goal.
2353
2354     Termination:
2355     The task terminates if the ball goes out of bounds or if the agent
2356         fails to kick the ball within a certain time frame.
2357     """
2358
2359     def __init__(self):
2360         super().__init__()
2361         self.ground_size = [1000.0, 1000.0, 10.0]
2362         self.ground_position = [0.0, 0.0, 0.0]
2363         self.ground_id = self.create_box(mass=0.0, half_extents=[self.
2364             ground_size[0] / 2, self.ground_size[1] / 2, self.ground_size
2365             [2] / 2], position=self.ground_position, color=[0.5, 0.5,
2366             0.5, 1.0])
2367         self._p.changeDynamics(bodyUniqueId=self.ground_id, linkIndex=-1,
2368             lateralFriction=0.8, restitution=0.5)
2369         self.ball_radius = 0.5
2370         self.ball_id = self.create_sphere(mass=1.0, radius=self.
2371             ball_radius, position=[0.0, 0.0, 0.0], color=[1.0, 0.0, 0.0,
2372             1.0])
2373         self.goal_post_height = 2.0
2374         self.goal_post_radius = 0.1
2375         self.goal_width = 3.0
2376         self.goal_position_init = [10.0, 0.0, self.ground_position[2] +
2377             self.ground_size[2] / 2 + self.goal_post_height / 2]
2378         self.goal_post_left_id = self.create_cylinder(mass=0.0, radius=
2379             self.goal_post_radius, height=self.goal_post_height, position
2380             =[self.goal_post_radius, height=self.goal_post_height, position
2381             =[self.goal_position_init[0], self.goal_position_init[1] -
2382                 self.goal_width / 2, self.goal_position_init[2]], color=[1.0,
2383                 1.0, 1.0, 1.0])
2384         self.goal_post_right_id = self.create_cylinder(mass=0.0, radius=
2385             self.goal_post_radius, height=self.goal_post_height, position
2386             =[self.goal_position_init[0], self.goal_position_init[1] +
2387                 self.goal_width / 2, self.goal_position_init[2]], color=[1.0,
2388                 1.0, 1.0, 1.0])

```

```

2376         self.goal_width / 2, self.goal_position_init[2]], color=[1.0,
2377             1.0, 1.0, 1.0])
2378     self.goal_amplitude = 2.0
2379     self.goal_period = 10.0
2380     self.robot_position_init = [self.ground_position[0], self.
2381         ground_position[1], self.ground_position[2] + self.
2382         ground_size[2] / 2 + self.robot.links['base'].position_init
2383         [2]]
2384
2384     def create_box(self, mass, half_extents, position, color):
2385         collision_shape_id = self._p.createCollisionShape(shapeType=self.
2386             _p.GEOM_BOX, halfExtents=half_extents)
2387         visual_shape_id = self._p.createVisualShape(shapeType=self._p.
2388             GEOM_BOX, halfExtents=half_extents, rgbaColor=color)
2389         return self._p.createMultiBody(baseMass=mass,
2390             baseCollisionShapeIndex=collision_shape_id,
2391             baseVisualShapeIndex=visual_shape_id, basePosition=position)
2392
2392     def create_sphere(self, mass, radius, position, color):
2393         collision_shape_id = self._p.createCollisionShape(shapeType=self.
2394             _p.GEOM_SPHERE, radius=radius)
2395         visual_shape_id = self._p.createVisualShape(shapeType=self._p.
2396             GEOM_SPHERE, radius=radius, rgbaColor=color)
2397         return self._p.createMultiBody(baseMass=mass,
2398             baseCollisionShapeIndex=collision_shape_id,
2399             baseVisualShapeIndex=visual_shape_id, basePosition=position)
2400
2400     def create_cylinder(self, mass, radius, height, position, color):
2401         collision_shape_id = self._p.createCollisionShape(shapeType=self.
2402             _p.GEOM_CYLINDER, radius=radius, height=height)
2403         visual_shape_id = self._p.createVisualShape(shapeType=self._p.
2404             GEOM_CYLINDER, radius=radius, length=height, rgbaColor=color)
2405         return self._p.createMultiBody(baseMass=mass,
2406             baseCollisionShapeIndex=collision_shape_id,
2407             baseVisualShapeIndex=visual_shape_id, basePosition=position)
2408
2408     def get_object_position(self, object_id):
2409         return np.asarray(self._p.getBasePositionAndOrientation(object_id
2410             )[0])
2411
2411     def get_distance_to_object(self, object_id):
2412         object_position = self.get_object_position(object_id)
2413         robot_position = self.robot.links['base'].position
2414         return np.linalg.norm(object_position[:2] - robot_position[:2])
2415
2415     def reset(self):
2416         observation = super().reset()
2417         self.time = 0.0
2418         self._p.resetBasePositionAndOrientation(self.robot.robot_id, self.
2419             .robot_position_init, self.robot.links['base'].
2420             orientation_init)
2421         self._p.resetBasePositionAndOrientation(self.ball_id, [self.
2422             robot_position_init[0] + 5.0, self.robot_position_init[1],
2423             self.ground_position[2] + self.ground_size[2] / 2 + self.
2424             ball_radius], [0.0, 0.0, 0.0, 1.0])
2425         return observation
2426
2426     def step(self, action):
2427         self.distance_to_ball = self.get_distance_to_object(self.ball_id)
2428         self.ball_position = self.get_object_position(self.ball_id)
2429         observation, reward, terminated, truncated, info = super().step(
2429             action)
2430         self.time += self.dt
2431         goal_y = self.goal_amplitude * np.sin(2 * np.pi * self.time /
2432             self.goal_period)

```

```

2430     self._p.resetBasePositionAndOrientation(self.goal_post_left_id, [
2431         self.goal_position_init[0], self.goal_position_init[1] - self.
2432             .goal_width / 2 + goal_y, self.goal_position_init[2]], [0.0,
2433             0.0, 0.0, 1.0])
2434     self._p.resetBasePositionAndOrientation(self.goal_post_right_id,
2435         [self.goal_position_init[0], self.goal_position_init[1] +
2436             self.goal_width / 2 + goal_y, self.goal_position_init[2]],
2437         [0.0, 0.0, 0.0, 1.0])
2438     return (observation, reward, terminated, truncated, info)
2439
2440     def get_task_rewards(self, action):
2441         new_distance_to_ball = self.get_distance_to_object(self.ball_id)
2442         new_ball_position = self.get_object_position(self.ball_id)
2443         survival = 1.0
2444         reach_ball = (self.distance_to_ball - new_distance_to_ball) /
2445             self.dt
2446         kick_ball = (new_ball_position[0] - self.ball_position[0]) / self.
2447             .dt
2448         goal_y = self.goal_amplitude * np.sin(2 * np.pi * self.time /
2449             self.goal_period)
2450         ball_in_goal = 0.0
2451         if self.goal_position_init[0] - self.goal_post_radius <
2452             new_ball_position[0] < self.goal_position_init[0] + self.
2453                 goal_post_radius and self.goal_position_init[1] - self.
2454                     goal_width / 2 + goal_y < new_ball_position[1] < self.
2455                         goal_position_init[1] + self.goal_width / 2 + goal_y:
2456                             ball_in_goal = 10.0
2457         return {'survival': survival, 'reach_ball': reach_ball,
2458                 'kick_ball': kick_ball, 'ball_in_goal': ball_in_goal}
2459
2460     def get_terminated(self, action):
2461         ball_position = self.get_object_position(self.ball_id)
2462         out_of_bounds = ball_position[0] < 0 or ball_position[0] > self.
2463             ground_size[0] or ball_position[1] < 0 or (ball_position[1] >
2464                 self.ground_size[1])
2465         return out_of_bounds
2466
2467     def get_success(self):
2468         ball_position = self.get_object_position(self.ball_id)
2469         goal_y = self.goal_amplitude * np.sin(2 * np.pi * self.time /
2470             self.goal_period)
2471         return self.goal_position_init[0] - self.goal_post_radius <
2472             ball_position[0] < self.goal_position_init[0] + self.
2473                 goal_post_radius and self.goal_position_init[1] - self.
2474                     goal_width / 2 + goal_y < ball_position[1] < self.
2475                         goal_position_init[1] + self.goal_width / 2 + goal_y
2476
2477

```

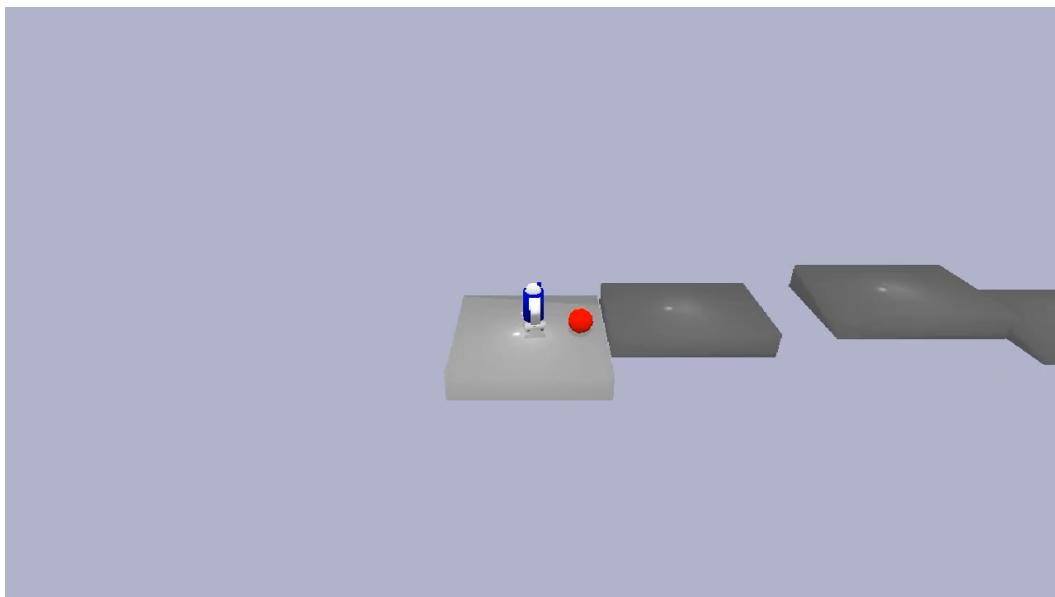
## Task 7

```

2471     import numpy as np
2472     from oped.envs.r2d2.base import R2D2Env
2473
2474     class Env(R2D2Env):
2475         """
2476             Cross a bridge with moving segments and kick a ball into a goal.
2477
2478             Description:
2479             - A start platform and an end platform (each 3 m in size and 0.5 m in
2480                 thickness) are placed 25 m apart.
2481             - The two platforms are connected by a bridge (2 m wide) divided into
2482                 6 segments of equal length (3 m each).
2483             - Each segment moves up and down independently with a sinusoidal
2484                 motion. The amplitude is 1 m and the period is 2 seconds, with
2485                 each segment offset in phase by 1/3 seconds from the previous one
2486             .
2487

```

2484  
2485  
2486  
2487  
2488  
2489  
2490  
2491  
2492  
2493  
2494  
2495  
2496  
2497  
2498  
2499  
2500  
2501  
2502  
2503



2504 Figure 15: Task 7 of the OMNI-EPIC run presented in Section 5.  
2505  
2506  
2507

- The segments have 1 m gaps between them.
- A spherical ball (0.5 m diameter) is placed on the start platform, 1 m in front of the robot’s initial position.
- A goal area (3 m wide and indicated by two posts) is placed on the end platform.

The robot is initialized on the start platform, facing the ball and bridge.

The task is for the robot to kick or push the ball across the dynamic bridge segments, navigating the gaps, to get the ball into the goal area on the other side. The robot must also successfully cross the bridge and reach the end platform.

#### Success:

The task is successfully completed when the ball reaches the goal area on the end platform and the robot also reaches the end platform intact.

#### Rewards:

- Reward for decreasing distance between robot and ball, encouraging the robot to approach and interact with the ball.
- Reward for increasing the forward progress of the ball across the bridge.
- Large reward for getting the ball into the goal area.
- Reward for robot staying on the bridge segments or platforms and avoiding falling.
- Reward for robot reaching the end platform.

#### Termination:

The episode terminates if the robot or ball falls off the bridge or platforms. It also terminates if the ball goes off the sides of the bridge or platforms.

....

```
2535     def __init__(self):
2536         super().__init__()
2537         self.platform_size = [3.0, 3.0, 0.5]
2538         self.platform_start_position = [0.0, 0.0, 0.0]
```

```

2538     self.platform_end_position = [self.platform_start_position[0] +
2539         25.0, self.platform_start_position[1], self.
2540         platform_start_position[2]]
2541     self.platform_start_id = self.create_box(mass=0.0, half_extents=[
2542         self.platform_size[0] / 2, self.platform_size[1] / 2, self.
2543         platform_size[2] / 2], position=self.platform_start_position,
2544         color=[0.8, 0.8, 0.8, 1.0])
2545     self.platform_end_id = self.create_box(mass=0.0, half_extents=[
2546         self.platform_size[0] / 2, self.platform_size[1] / 2, self.
2547         platform_size[2] / 2], position=self.platform_end_position,
2548         color=[0.8, 0.8, 0.8, 1.0])
2549     self.bridge_length = self.platform_end_position[0] - self.
2550         platform_start_position[0] - self.platform_size[0]
2551     self.bridge_width = 2.0
2552     self.num_segments = 6
2553     self.segment_length = 3.0
2554     self.gap_length = 1.0
2555     self.segment_amplitude = 1.0
2556     self.segment_period = 2.0
2557     self.segment_phase_offset = 1.0 / 3.0
2558     self.segment_ids = []
2559     for i in range(self.num_segments):
2560         segment_id = self.create_box(mass=0.0, half_extents=[self.
2561             segment_length / 2, self.bridge_width / 2, self.
2562             platform_size[2] / 2], position=[self.
2563             platform_start_position[0] + self.platform_size[0] / 2 +
2564             self.segment_length / 2 + i * (self.segment_length + self.
2565             .gap_length), self.platform_start_position[1], self.
2566             platform_start_position[2]], color=[0.5, 0.5, 0.5, 1.0])
2567         self._p.changeDynamics(bodyUniqueId=segment_id, linkIndex=-1,
2568             lateralFriction=0.8, restitution=0.5)
2569         self.segment_ids.append(segment_id)
2570     self.ball_radius = 0.25
2571     self.ball_position_init = [self.platform_start_position[0] + 1.0,
2572         self.platform_start_position[1], self.
2573         platform_start_position[2] + self.platform_size[2] / 2 + self
2574         .ball_radius]
2575     self.ball_id = self.create_sphere(mass=1.0, radius=self.
2576         ball_radius, position=self.ball_position_init, color=[1.0,
2577         0.0, 0.0, 1.0])
2578     self.goal_width = 3.0
2579     self.goal_position = [self.platform_end_position[0], self.
2580         platform_end_position[1], self.platform_end_position[2] +
2581         self.platform_size[2] / 2 + self.ball_radius]
2582     self.goal_left_post_id = self.create_cylinder(mass=0.0, radius
2583         =0.1, height=1.0, position=[self.goal_position[0], self.
2584         goal_position[1] - self.goal_width / 2, self.goal_position
2585         [2]], color=[0.0, 1.0, 0.0, 1.0])
2586     self.goal_right_post_id = self.create_cylinder(mass=0.0, radius
2587         =0.1, height=1.0, position=[self.goal_position[0], self.
2588         goal_position[1] + self.goal_width / 2, self.goal_position
2589         [2]], color=[0.0, 1.0, 0.0, 1.0])
2590
2591     def create_box(self, mass, half_extents, position, color):
2592         collision_shape_id = self._p.createCollisionShape(shapeType=self.
2593             _p.GEOM_BOX, halfExtents=half_extents)
2594         visual_shape_id = self._p.createVisualShape(shapeType=self._p.
2595             GEOM_BOX, halfExtents=half_extents, rgbaColor=color)
2596         return self._p.createMultiBody(baseMass=mass,
2597             baseCollisionShapeIndex=collision_shape_id,
2598             baseVisualShapeIndex=visual_shape_id, basePosition=position)
2599
2600     def create_sphere(self, mass, radius, position, color):
2601         collision_shape_id = self._p.createCollisionShape(shapeType=self.
2602             _p.GEOM_SPHERE, radius=radius)

```

```

2592     visual_shape_id = self._p.createVisualShape(shapeType=self._p.
2593         GEOM_SPHERE, radius=radius, rgbaColor=color)
2594     return self._p.createMultiBody(baseMass=mass,
2595         baseCollisionShapeIndex=collision_shape_id,
2596         baseVisualShapeIndex=visual_shape_id, basePosition=position)
2597
2598     def create_cylinder(self, mass, radius, height, position, color):
2599         collision_shape_id = self._p.createCollisionShape(shapeType=self.
2600             _p.GEOM_CYLINDER, radius=radius, height=height)
2601         visual_shape_id = self._p.createVisualShape(shapeType=self._p.
2602             GEOM_CYLINDER, radius=radius, length=height, rgbaColor=color)
2603         return self._p.createMultiBody(baseMass=mass,
2604             baseCollisionShapeIndex=collision_shape_id,
2605             baseVisualShapeIndex=visual_shape_id, basePosition=position)
2606
2607     def get_object_position(self, object_id):
2608         return np.asarray(self._p.getBasePositionAndOrientation(object_id
2609             )[0])
2610
2611     def get_distance_to_object(self, object_id):
2612         object_position = self.get_object_position(object_id)
2613         robot_position = self.robot.links['base'].position
2614         return np.linalg.norm(object_position[:2] - robot_position[:2])
2615
2616     def reset(self):
2617         observation = super().reset()
2618         self.time = 0.0
2619         self._p.resetBasePositionAndOrientation(self.ball_id, self.
2620             ball_position_init, [0.0, 0.0, 0.0, 1.0])
2621         self._p.resetBasePositionAndOrientation(self.robot.robot_id, [
2622             self.platform_start_position[0], self.platform_start_position
2623             [1], self.platform_start_position[2] + self.platform_size[2]
2624             / 2 + self.robot.links['base'].position_init[2]], self.robot.
2625             links['base'].orientation_init)
2626         return observation
2627
2628     def step(self, action):
2629         self.distance_to_ball = self.get_distance_to_object(self.ball_id)
2630         self.ball_position = self.get_object_position(self.ball_id)
2631         self.distance_to_platform_end = self.get_distance_to_object(self.
2632             platform_end_id)
2633         observation, reward, terminated, truncated, info = super().step(
2634             action)
2635         self.time += self.dt
2636         for i, segment_id in enumerate(self.segment_ids):
2637             segment_position = self.get_object_position(segment_id)
2638             new_segment_position = [segment_position[0], segment_position
2639                 [1], self.platform_start_position[2] + self.
2640                 segment_amplitude * np.sin(2 * np.pi * (self.time + i *
2641                     self.segment_phase_offset) / self.segment_period)]
2642             self._p.resetBasePositionAndOrientation(segment_id,
2643                 new_segment_position, [0.0, 0.0, 0.0, 1.0])
2644         return (observation, reward, terminated, truncated, info)
2645
2646     def get_task_rewards(self, action):
2647         new_distance_to_ball = self.get_distance_to_object(self.ball_id)
2648         new_ball_position = self.get_object_position(self.ball_id)
2649         new_distance_to_platform_end = self.get_distance_to_object(self.
2650             platform_end_id)
2651         approach_ball = (self.distance_to_ball - new_distance_to_ball) /
2652             self.dt
2653         ball_forward_velocity = (new_ball_position[0] - self.
2654             ball_position[0]) / self.dt
2655         ball_in_goal = 10.0 if self.goal_position[1] - self.goal_width /
2656             2 < new_ball_position[1] < self.goal_position[1] + self.

```

```

2646         goal_width / 2 and new_ball_position[0] > self.goal_position
2647         [0] else 0.0
2648     on_bridge_or_platforms = 1.0 if self.robot.links['base'].position
2649         [2] > self.platform_start_position[2] + self.platform_size[2]
2650         / 2 else -1.0
2651     reach_platform_end = (self.distance_to_platform_end -
2652         new_distance_to_platform_end) / self.dt
2653     return {'approach_ball': approach_ball, 'ball_forward_velocity':
2654         ball_forward_velocity, 'ball_in_goal': ball_in_goal, '
2655         on_bridge_or_platforms': on_bridge_or_platforms, '
2656         reach_platform_end': reach_platform_end}

2657 def get_terminated(self, action):
2658     is_robot_on_bridge_or_platforms = self.robot.links['base'].
2659         position[2] > self.platform_start_position[2]
2660     is_ball_on_bridge_or_platforms = self.get_object_position(self.
2661         ball_id)[2] > self.platform_start_position[2]
2662     is_ball_off_sides = np.abs(self.get_object_position(self.ball_id)
2663         [1]) > self.bridge_width / 2 + self.platform_size[1] / 2
2664     return not is_robot_on_bridge_or_platforms or not
2665         is_ball_on_bridge_or_platforms or is_ball_off_sides

2666 def get_success(self):
2667     ball_position = self.get_object_position(self.ball_id)
2668     is_ball_in_goal = self.goal_position[1] - self.goal_width / 2 <
2669         ball_position[1] < self.goal_position[1] + self.goal_width /
2670         2 and ball_position[0] > self.goal_position[0]
2671     is_robot_on_platform_end = self.get_distance_to_object(self.
2672         platform_end_id) < self.platform_size[2] / 2
2673     return is_ball_in_goal and is_robot_on_platform_end

```

### Task 8

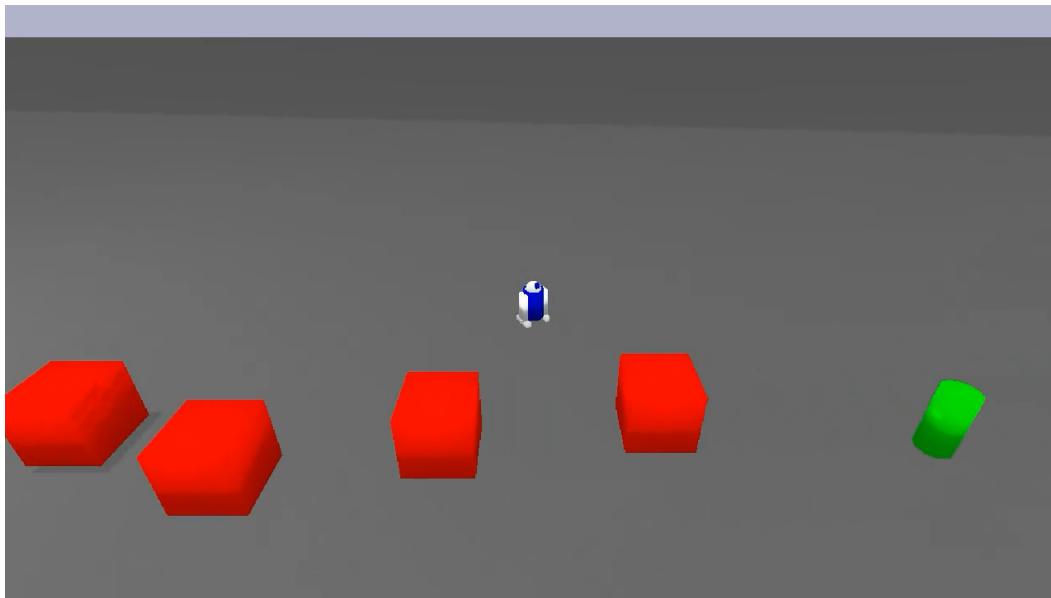


Figure 16: Task 8 of the OMNI-EPIC run presented in Section 5.

```

2694 import numpy as np
2695 from oped.envs.r2d2.base import R2D2Env
2696
2697 class Env(R2D2Env):
2698     """
2699

```

2700           Navigate a dynamic obstacle course to reach a target.  
 2701  
 2702           **Description:**  
 2703           – The environment consists of a large flat ground measuring 1000 x  
               1000 x 10 meters.  
 2704           – The robot starts at a fixed position on the ground, facing the  
               positive x-axis.  
 2705           – The target is a cylindrical object (0.5 meters in diameter and 1  
               meter in height) placed 20 meters directly in front of the robot'  
               s starting position.  
 2706           – The obstacle course consists of 5 dynamic obstacles, each moving in  
               a predictable pattern:  
 2707           – Obstacle 1: A box (1 x 1 x 1 meters) moving side-to-side (along  
               the y-axis) with an amplitude of 2 meters and a period of 4  
               seconds. It is placed 5 meters in front of the robot's starting  
               position.  
 2708           – Obstacle 2: A box (1 x 1 x 1 meters) moving up and down (along  
               the z-axis) with an amplitude of 1 meter and a period of 3  
               seconds. It is placed 8 meters in front of the robot's starting  
               position.  
 2709           – Obstacle 3: A box (1 x 1 x 1 meters) rotating around its center  
               with a radius of 1 meter and a period of 5 seconds. It is  
               placed 11 meters in front of the robot's starting position.  
 2710           – Obstacle 4: A box (1 x 1 x 1 meters) moving diagonally (along  
               both x and y axes) with an amplitude of 1 meter and a period of  
               6 seconds. It is placed 14 meters in front of the robot's  
               starting position.  
 2711           – Obstacle 5: A box (1 x 1 x 1 meters) moving in a circular pattern  
               (along the x-y plane) with a radius of 1 meter and a period of  
               7 seconds. It is placed 17 meters in front of the robot's  
               starting position.  
 2712           – The task of the robot is to navigate through the dynamic obstacle  
               course and reach the target as quickly as possible.  
 2713  
 2714           **Success:**  
 2715           The task is successfully completed when the robot reaches the target.  
 2716  
 2717           **Rewards:**  
 2718           To help the robot complete the task:  
 2719           – The robot receives a reward for each time step it remains in the  
               environment, encouraging steady progress.  
 2720           – The robot is rewarded based on how much it reduces the distance to  
               the target, incentivizing swift movement towards the goal.  
 2721           – The robot is penalized for colliding with any obstacles.  
 2722  
 2723           **Termination:**  
 2724           The task terminates immediately if the robot collides with any  
               obstacle or reaches the target.  
 2725           \*\*\*  
 2726  
 2727           **def \_\_init\_\_(self):**  
 2728            super().\_\_init\_\_()  
 2729            self.ground\_size = [1000.0, 1000.0, 10.0]  
 2730            self.ground\_position = [0.0, 0.0, 0.0]  
 2731            self.ground\_id = self.create\_box(mass=0.0, half\_extents=[self.  
 2732                ground\_size[0] / 2, self.ground\_size[1] / 2, self.ground\_size  
 2733                [2] / 2], position=self.ground\_position, color=[0.5, 0.5,  
 2734                0.5, 1.0])  
 2735            self.\_p.changeDynamics(bodyUniqueId=self.ground\_id, linkIndex=-1,  
 2736                lateralFriction=0.8, restitution=0.5)  
 2737            self.target\_radius = 0.25  
 2738            self.target\_height = 1.0  
 2739            self.target\_position = [20.0, 0.0, self.ground\_position[2] + self  
 2740                .ground\_size[2] / 2 + self.target\_height / 2]

```

2754     self.target_id = self.create_cylinder(mass=0.0, radius=self.
2755         target_radius, height=self.target_height, position=self.
2756         target_position, color=[0.0, 1.0, 0.0, 1.0])
2757     self.obstacle_size = [1.0, 1.0, 1.0]
2758     self.obstacle_1_position_init = [5.0, 0.0, self.ground_position
2759         [2] + self.ground_size[2] / 2 + self.obstacle_size[2] / 2]
2760     self.obstacle_2_position_init = [8.0, 0.0, self.ground_position
2761         [2] + self.ground_size[2] / 2 + self.obstacle_size[2] / 2]
2762     self.obstacle_3_position_init = [11.0, 0.0, self.ground_position
2763         [2] + self.ground_size[2] / 2 + self.obstacle_size[2] / 2]
2764     self.obstacle_4_position_init = [14.0, 0.0, self.ground_position
2765         [2] + self.ground_size[2] / 2 + self.obstacle_size[2] / 2]
2766     self.obstacle_5_position_init = [17.0, 0.0, self.ground_position
2767         [2] + self.ground_size[2] / 2 + self.obstacle_size[2] / 2]
2768     self.obstacle_1_id = self.create_box(mass=0.0, half_extents=[self.
2769         .obstacle_size[0] / 2, self.obstacle_size[1] / 2, self.
2770         .obstacle_size[2] / 2], position=self.obstacle_1_position_init
2771         , color=[1.0, 0.0, 0.0, 1.0])
2772     self.obstacle_2_id = self.create_box(mass=0.0, half_extents=[self.
2773         .obstacle_size[0] / 2, self.obstacle_size[1] / 2, self.
2774         .obstacle_size[2] / 2], position=self.obstacle_2_position_init
2775         , color=[1.0, 0.0, 0.0, 1.0])
2776     self.obstacle_3_id = self.create_box(mass=0.0, half_extents=[self.
2777         .obstacle_size[0] / 2, self.obstacle_size[1] / 2, self.
2778         .obstacle_size[2] / 2], position=self.obstacle_3_position_init
2779         , color=[1.0, 0.0, 0.0, 1.0])
2780     self.obstacle_4_id = self.create_box(mass=0.0, half_extents=[self.
2781         .obstacle_size[0] / 2, self.obstacle_size[1] / 2, self.
2782         .obstacle_size[2] / 2], position=self.obstacle_4_position_init
2783         , color=[1.0, 0.0, 0.0, 1.0])
2784     self.obstacle_5_id = self.create_box(mass=0.0, half_extents=[self.
2785         .obstacle_size[0] / 2, self.obstacle_size[1] / 2, self.
2786         .obstacle_size[2] / 2], position=self.obstacle_5_position_init
2787         , color=[1.0, 0.0, 0.0, 1.0])
2788     self.robot_position_init = [0.0, 0.0, self.ground_position[2] +
2789         self.ground_size[2] / 2 + self.robot.links['base'].
2790         position_init[2]]
2791     self.robot_orientation_init = self._p.getQuaternionFromEuler
2792         ([0.0, 0.0, 0.0])
2793
2794     def create_box(self, mass, half_extents, position, color):
2795         collision_shape_id = self._p.createCollisionShape(shapeType=self.
2796             ._p.GEOM_BOX, halfExtents=half_extents)
2797         visual_shape_id = self._p.createVisualShape(shapeType=self._p.
2798             GEOM_BOX, halfExtents=half_extents, rgbaColor=color)
2799         return self._p.createMultiBody(baseMass=mass,
2800             baseCollisionShapeIndex=collision_shape_id,
2801             baseVisualShapeIndex=visual_shape_id, basePosition=position)
2802
2803     def create_cylinder(self, mass, radius, height, position, color):
2804         collision_shape_id = self._p.createCollisionShape(shapeType=self.
2805             ._p.GEOM_CYLINDER, radius=radius, height=height)
2806         visual_shape_id = self._p.createVisualShape(shapeType=self._p.
2807             GEOM_CYLINDER, radius=radius, length=height, rgbaColor=color)
2808         return self._p.createMultiBody(baseMass=mass,
2809             baseCollisionShapeIndex=collision_shape_id,
2810             baseVisualShapeIndex=visual_shape_id, basePosition=position)
2811
2812     def get_object_position(self, object_id):
2813         return np.asarray(self._p.getBasePositionAndOrientation(object_id
2814             )[0])
2815
2816     def get_distance_to_object(self, object_id):
2817         object_position = self.get_object_position(object_id)
2818         robot_position = self.robot.links['base'].position

```

```

2808     return np.linalg.norm(object_position[:2] - robot_position[:2])
2809
2810     def reset(self):
2811         observation = super().reset()
2812         self.time = 0.0
2813         self._p.resetBasePositionAndOrientation(self.robot.robot_id, self.
2814             .robot_position_init, self.robot_orientation_init)
2815         return observation
2816
2817     def step(self, action):
2818         self.distance_to_target = self.get_distance_to_object(self.
2819             .target_id)
2820         observation, reward, terminated, truncated, info = super().step(
2821             action)
2822         self.time += self.dt
2823         obstacle_1_new_y = self.obstacle_1_position_init[1] + 2.0 * np.
2824             sin(2 * np.pi * self.time / 4.0)
2825         self._p.resetBasePositionAndOrientation(self.obstacle_1_id, [self.
2826             .obstacle_1_position_init[0], obstacle_1_new_y, self.
2827             .obstacle_1_position_init[2]], [0.0, 0.0, 0.0, 1.0])
2828         obstacle_2_new_z = self.obstacle_2_position_init[2] + 1.0 * np.
2829             sin(2 * np.pi * self.time / 3.0)
2830         self._p.resetBasePositionAndOrientation(self.obstacle_2_id, [self.
2831             .obstacle_2_position_init[0], self.obstacle_2_position_init
2832                 [1], obstacle_2_new_z], [0.0, 0.0, 0.0, 1.0])
2833         obstacle_3_new_x = self.obstacle_3_position_init[0] + 1.0 * np.
2834             cos(2 * np.pi * self.time / 5.0)
2835         obstacle_3_new_y = self.obstacle_3_position_init[1] + 1.0 * np.
2836             sin(2 * np.pi * self.time / 5.0)
2837         self._p.resetBasePositionAndOrientation(self.obstacle_3_id, [
2838             obstacle_3_new_x, obstacle_3_new_y, self.
2839             .obstacle_3_position_init[2]], [0.0, 0.0, 0.0, 1.0])
2840         obstacle_4_new_x = self.obstacle_4_position_init[0] + 1.0 * np.
2841             cos(2 * np.pi * self.time / 6.0)
2842         obstacle_4_new_y = self.obstacle_4_position_init[1] + 1.0 * np.
2843             sin(2 * np.pi * self.time / 6.0)
2844         self._p.resetBasePositionAndOrientation(self.obstacle_4_id, [
2845             obstacle_4_new_x, obstacle_4_new_y, self.
2846             .obstacle_4_position_init[2]], [0.0, 0.0, 0.0, 1.0])
2847         obstacle_5_new_x = self.obstacle_5_position_init[0] + 1.0 * np.
2848             cos(2 * np.pi * self.time / 7.0)
2849         obstacle_5_new_y = self.obstacle_5_position_init[1] + 1.0 * np.
2850             sin(2 * np.pi * self.time / 7.0)
2851         self._p.resetBasePositionAndOrientation(self.obstacle_5_id, [
2852             obstacle_5_new_x, obstacle_5_new_y, self.
2853             .obstacle_5_position_init[2]], [0.0, 0.0, 0.0, 1.0])
2854         return (observation, reward, terminated, truncated, info)
2855
2856     def get_task_rewards(self, action):
2857         new_distance_to_target = self.get_distance_to_object(self.
2858             .target_id)
2859         survival = 1.0
2860         reach_target = (self.distance_to_target - new_distance_to_target)
2861             / self.dt
2862         return {'survival': survival, 'reach_target': reach_target}
2863
2864     def get_terminated(self, action):
2865         collision_obstacle_1 = len(self._p.getContactPoints(bodyA=self.
2866             .robot.robot_id, bodyB=self.obstacle_1_id)) > 0
2867         collision_obstacle_2 = len(self._p.getContactPoints(bodyA=self.
2868             .robot.robot_id, bodyB=self.obstacle_2_id)) > 0
2869         collision_obstacle_3 = len(self._p.getContactPoints(bodyA=self.
2870             .robot.robot_id, bodyB=self.obstacle_3_id)) > 0
2871         collision_obstacle_4 = len(self._p.getContactPoints(bodyA=self.
2872             .robot.robot_id, bodyB=self.obstacle_4_id)) > 0

```

```

2862     collision_obstacle_5 = len(self._p.getContactPoints(bodyA=self.
2863         .robot.robot_id, bodyB=self.obstacle_5_id)) > 0
2864     collision_target = len(self._p.getContactPoints(bodyA=self.robot.
2865         robot_id, bodyB=self.target_id)) > 0
2866     return collision_obstacle_1 or collision_obstacle_2 or
2867         collision_obstacle_3 or collision_obstacle_4 or
2868         collision_obstacle_5 or collision_target
2869
2870     def get_success(self):
2871         contact_points_target = self._p.getContactPoints(bodyA=self.robot.
2872             .robot_id, bodyB=self.target_id)
2873         return len(contact_points_target) > 0
2874
2875
2876
2877
2878
2879
2880
2881
2882
2883
2884
2885
2886
2887
2888
2889
2890
2891
2892
2893
2894
2895
2896
2897
2898
2899
2900
2901
2902
2903
2904
2905
2906
2907
2908
2909
2910
2911
2912
2913
2914
2915

```

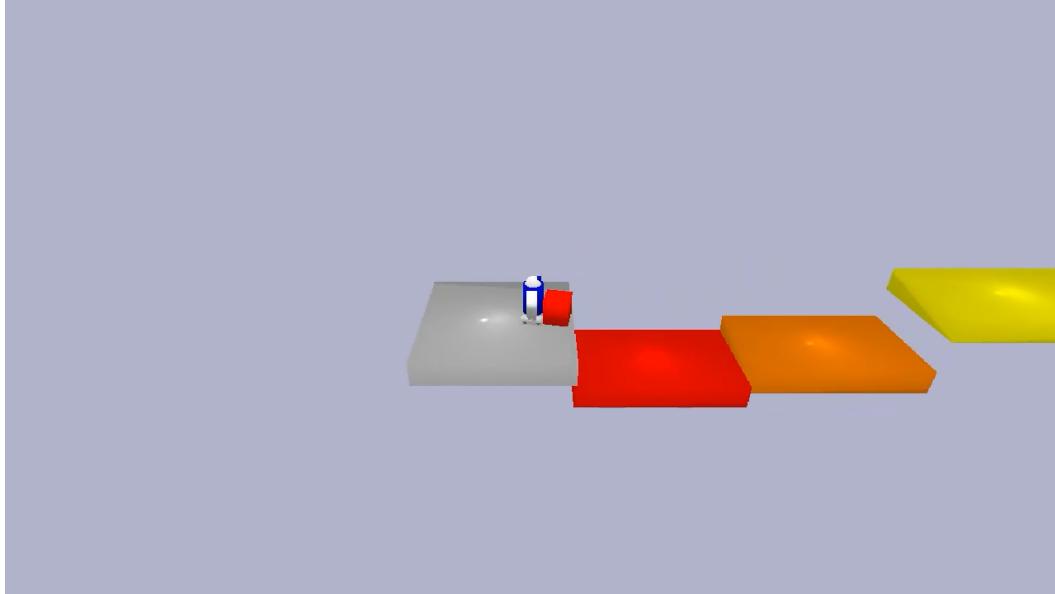
**Task 9**

Figure 17: Task 9 of the OMNI-EPIC run presented in Section 5.

```

import numpy as np
from oped.envs.r2d2.base import R2D2Env

class Env(R2D2Env):
    """
        Push a box to a target location on a dynamic platform.

    Description:
        - The environment consists of a start platform and an end platform (each 3 m in size and 0.5 m in thickness) placed 20 m apart.
        - The two platforms are connected by a dynamic bridge (2 m wide) divided into 5 segments of equal length (3 m each).
        - Each segment moves up and down independently with a sinusoidal motion. The amplitude is 1 m and the period is 2 seconds, with each segment offset in phase by 0.4 seconds from the previous one .
        - A box (0.5 m in size) is placed on the start platform, 1 m in front of the robot's initial position.
        - The robot is initialized on the start platform, facing the box and bridge.
        - The task of the robot is to push the box across the dynamic bridge segments to reach a target location on the end platform.
    """

```

```

2916 Success:
2917 The task is successfully completed when the box reaches the target
2918 location on the end platform.
2919
2920 Rewards:
2921 - Reward for decreasing the distance between the robot and the box,
2922   encouraging the robot to approach and interact with the box.
2923 - Reward for increasing the forward progress of the box across the
2924   bridge.
2925 - Large reward for getting the box to the target location on the end
2926   platform.
2927 - Reward for the robot staying on the bridge segments or platforms
2928   and avoiding falling.
2929 - Reward for the robot reaching the end platform.
2930
2931 Termination:
2932 The episode terminates if the robot or box falls off the bridge or
2933 platforms.
2934 """
2935
2936 def __init__(self):
2937     super().__init__()
2938     self.platform_size = [3.0, 3.0, 0.5]
2939     self.platform_start_position = [0.0, 0.0, 0.0]
2940     self.platform_end_position = [self.platform_start_position[0] +
2941                                   20.0, self.platform_start_position[1], self.
2942                                   platform_start_position[2]]
2943     self.platform_start_id = self.create_box(mass=0.0, half_extents=[
2944         self.platform_size[0] / 2, self.platform_size[1] / 2, self.
2945         platform_size[2] / 2], position=self.platform_start_position,
2946         color=[0.8, 0.8, 0.8, 1.0])
2947     self.platform_end_id = self.create_box(mass=0.0, half_extents=[[
2948         self.platform_size[0] / 2, self.platform_size[1] / 2, self.
2949         platform_size[2] / 2], position=self.platform_end_position,
2950         color=[0.8, 0.8, 0.8, 1.0]])
2951     self.bridge_length = self.platform_end_position[0] - self.
2952     platform_start_position[0] - self.platform_size[0]
2953     self.bridge_width = 2.0
2954     self.num_segments = 5
2955     self.segment_length = self.bridge_length / self.num_segments
2956     self.segment_amplitude = 1.0
2957     self.segment_period = 2.0
2958     self.segment_phase_offset = 0.4
2959     segment_colors = [[1.0, 0.0, 0.0, 1.0], [1.0, 0.5, 0.0, 1.0],
2960                       [1.0, 1.0, 0.0, 1.0], [0.0, 1.0, 0.0, 1.0], [0.0, 0.0,
2961                         1.0]]
2962     self.segment_ids = []
2963     for i in range(self.num_segments):
2964         segment_id = self.create_box(mass=0.0, half_extents=[self.
2965             segment_length / 2, self.bridge_width / 2, self.
2966             platform_size[2] / 2], position=[self.
2967             platform_start_position[0] + self.platform_size[0] / 2 +
2968             self.segment_length / 2 + i * self.segment_length, self.
2969             platform_start_position[1], self.platform_start_position
2970             [2]], color=segment_colors[i])
2971         self._p.changeDynamics(bodyUniqueId=segment_id, linkIndex=-1,
2972                               lateralFriction=0.8, restitution=0.5)
2973         self.segment_ids.append(segment_id)
2974     self.box_size = [0.5, 0.5, 0.5]
2975     self.box_position_init = [self.platform_start_position[0] + 1.0,
2976                               self.platform_start_position[1], self.platform_start_position
2977                               [2] + self.platform_size[2] / 2 + self.box_size[2] / 2]
2978     self.box_id = self.create_box(mass=1.0, half_extents=[self.
2979       box_size[0] / 2, self.box_size[1] / 2, self.box_size[2] / 2],
2980       position=self.box_position_init, color=[1.0, 0.0, 0.0, 1.0])

```

```

2970
2971     self.target_position = [self.platform_end_position[0], self.
2972         platform_end_position[1], self.platform_end_position[2] +
2973             self.platform_size[2] / 2 + self.box_size[2] / 2]
2974
2975     def create_box(self, mass, half_extents, position, color):
2976         collision_shape_id = self._p.createCollisionShape(shapeType=self.
2977             _p.GEOM_BOX, halfExtents=half_extents)
2978         visual_shape_id = self._p.createVisualShape(shapeType=self._p.
2979             GEOM_BOX, halfExtents=half_extents, rgbaColor=color)
2980         return self._p.createMultiBody(baseMass=mass,
2981             baseCollisionShapeIndex=collision_shape_id,
2982             baseVisualShapeIndex=visual_shape_id, basePosition=position)
2983
2984     def get_object_position(self, object_id):
2985         return np.asarray(self._p.getBasePositionAndOrientation(object_id
2986             )[0])
2987
2988     def get_distance_to_object(self, object_id):
2989         object_position = self.get_object_position(object_id)
2990         robot_position = self.robot.links['base'].position
2991         return np.linalg.norm(object_position[:2] - robot_position[:2])
2992
2993     def reset(self):
2994         observation = super().reset()
2995         self.time = 0.0
2996         self._p.resetBasePositionAndOrientation(self.robot.robot_id, [
2997             self.platform_start_position[0], self.platform_start_position
2998             [1], self.platform_start_position[2] + self.platform_size[2]
2999             / 2 + self.robot.links['base'].position_init[2]], self.robot.
3000             links['base'].orientation_init)
3001         self._p.resetBasePositionAndOrientation(self.box_id, self.
3002             box_position_init, [0.0, 0.0, 0.0, 1.0])
3003         return observation
3004
3005     def step(self, action):
3006         self.distance_to_box = self.get_distance_to_object(self.box_id)
3007         self.box_position = self.get_object_position(self.box_id)
3008         observation, reward, terminated, truncated, info = super().step(
3009             action)
3010         self.time += self.dt
3011         for i, segment_id in enumerate(self.segment_ids):
3012             segment_position = self.get_object_position(segment_id)
3013             new_segment_position = [segment_position[0], segment_position
3014                 [1], self.platform_start_position[2] + self.
3015                 segment_amplitude * np.sin(2 * np.pi * (self.time + i *
3016                     self.segment_phase_offset) / self.segment_period)]
3017             self._p.resetBasePositionAndOrientation(segment_id,
3018                 new_segment_position, [0.0, 0.0, 0.0, 1.0])
3019         return (observation, reward, terminated, truncated, info)
3020
3021     def get_task_rewards(self, action):
3022         new_distance_to_box = self.get_distance_to_object(self.box_id)
3023         new_box_position = self.get_object_position(self.box_id)
3024         approach_box = (self.distance_to_box - new_distance_to_box) /
3025             self.dt
3026         forward_progress_box = (new_box_position[0] - self.box_position
3027             [0]) / self.dt
3028         on_bridge_or_platforms = 1.0 if self.robot.links['base'].position
3029             [2] > self.platform_start_position[2] + self.platform_size[2]
3030             / 2 else -1.0
3031         reach_end_platform = 1.0 if self.robot.links['base'].position[0]
3032             >= self.platform_end_position[0] else 0.0
3033         reach_target_location = 10.0 if np.linalg.norm(new_box_position
3034             [:2] - self.target_position[:2]) < self.box_size[0] / 2 else
3035             0.0

```

```
3024     return {'approach_box': approach_box, 'forward_progress_box':  
3025         forward_progress_box, 'on_bridge_or_platforms':  
3026         on_bridge_or_platforms, 'reach_end_platform':  
3027         reach_end_platform, 'reach_target_location':  
3028         reach_target_location}  
  
3029 def get_terminated(self, action):  
3030     robot_fell = self.robot.links['base'].position[2] < self.  
3031         platform_start_position[2]  
3032     box_fell = self.get_object_position(self.box_id)[2] < self.  
3033         platform_start_position[2]  
3034     return robot_fell or box_fell  
  
3035 def get_success(self):  
3036     box_position = self.get_object_position(self.box_id)  
3037     return np.linalg.norm(box_position[:2] - self.target_position  
3038         [:2]) < self.box_size[0] / 2
```

## Task 10

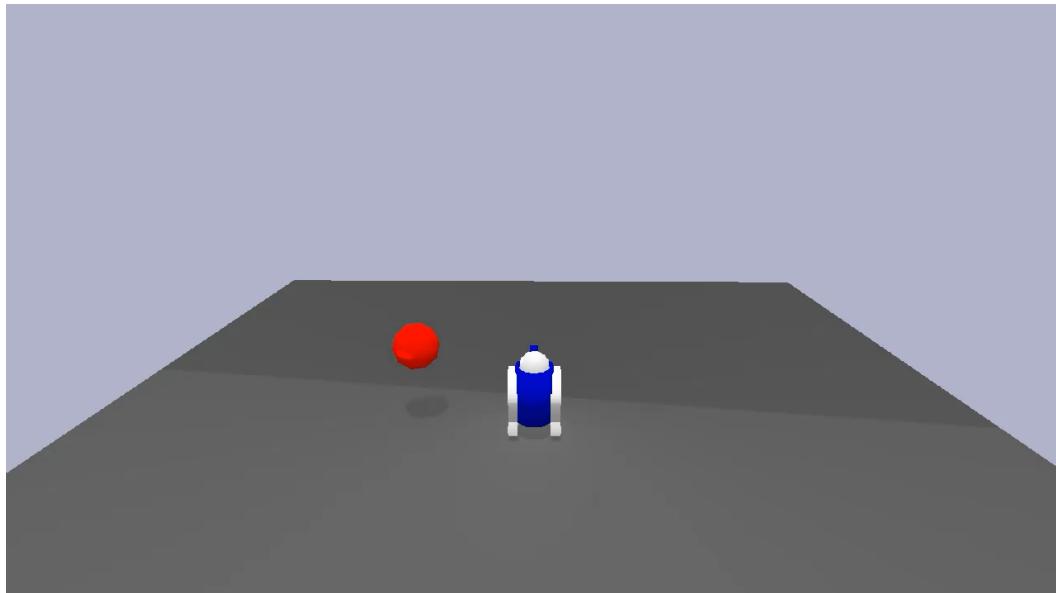


Figure 18: Task 10 of the OMNI-EPIC run presented in Section 5.

```
3064 import numpy as np
3065 from oped.envs.r2d2.base import R2D2Env
3066
3067 class Env(R2D2Env):
3068     """
3069     Dodgeball Game
3070
3071     Description:
3072     - The robot spawns in a square arena measuring 10 meters by 10 meters
3073     .
3074     - Balls with a radius of 0.25 meters spawn randomly within the arena
3075         at a height of 1 meter and are launched toward the robot with an
3076         initial velocity of 5 meters per second.
3077     - The balls spawn at a rate of one ball every 2 seconds.
3078     - The robot's task is to dodge the balls for as long as possible.
3079
3080     Success:
```

```

3078 The task is successfully completed if the robot survives for a
3079 predefined duration of 60 seconds without being hit by any balls.
3080
3081 Rewards:
3082 - The robot receives a reward for each time step it remains standing
3083 and avoids being hit by a ball, encouraging it to dodge
3084 effectively.
3085 - A small survival reward is given for each second the robot remains
3086 in the game, incentivizing prolonged survival.
3087 - A penalty is applied if the robot is hit by a ball, encouraging it
3088 to avoid collisions.
3089
3090 Termination:
3091 - The task terminates immediately if the robot is hit by a ball.
3092 - The task also terminates if the robot survives for 60 seconds
3093     without being hit by any balls.
3094 """
3095
3096 def __init__(self):
3097     super().__init__()
3098     self.arena_size = [10.0, 10.0]
3099     self.arena_position = [0.0, 0.0, 0.0]
3100     self.arena_id = self.create_box(mass=0.0, half_extents=[self.
3101         arena_size[0] / 2, self.arena_size[1] / 2, 0.1], position=
3102         self.arena_position, color=[0.5, 0.5, 0.5, 1.0])
3103     self._p.changeDynamics(bodyUniqueId=self.arena_id, linkIndex=-1,
3104         lateralFriction=0.8, restitution=0.5)
3105     self.ball_radius = 0.25
3106     self.ball_velocity = 5.0
3107     self.ball_spawn_interval = 2.0
3108     self.ball_ids = []
3109     self.robot_position_init = [0.0, 0.0, self.arena_position[2] +
3110         0.1 + self.robot.links['base'].position_init[2]]
3111     self.robot_orientation_init = self._p.getQuaternionFromEuler
3112         ([0.0, 0.0, 0.0])
3113     self.survival_duration = 16.0
3114
3115     def create_box(self, mass, half_extents, position, color):
3116         collision_shape_id = self._p.createCollisionShape(shapeType=self.
3117             _p.GEOM_BOX, halfExtents=half_extents)
3118         visual_shape_id = self._p.createVisualShape(shapeType=self._p.
3119             GEOM_BOX, halfExtents=half_extents, rgbaColor=color)
3120         return self._p.createMultiBody(baseMass=mass,
3121             baseCollisionShapeIndex=collision_shape_id,
3122             baseVisualShapeIndex=visual_shape_id, basePosition=position)
3123
3124     def create_sphere(self, mass, radius, position, velocity, color):
3125         collision_shape_id = self._p.createCollisionShape(shapeType=self.
3126             _p.GEOM_SPHERE, radius=radius)
3127         visual_shape_id = self._p.createVisualShape(shapeType=self._p.
3128             GEOM_SPHERE, radius=radius, rgbaColor=color)
3129         ball_id = self._p.createMultiBody(baseMass=mass,
3130             baseCollisionShapeIndex=collision_shape_id,
3131             baseVisualShapeIndex=visual_shape_id, basePosition=position)
3132         self._p.resetBaseVelocity(objectUniqueId=ball_id, linearVelocity=
3133             velocity)
3134         return ball_id
3135
3136     def reset(self):
3137         observation = super().reset()
3138         self.time = 0.0
3139         self._p.resetBasePositionAndOrientation(self.robot.robot_id, self
3140             .robot_position_init, self.robot_orientation_init)
3141         for ball_id in self.ball_ids:
3142             self._p.removeBody(ball_id)

```

```

3132         self.ball_ids = []
3133         return observation
3134
3135     def step(self, action):
3136         self.hit_by_ball = False
3137         observation, reward, terminated, truncated, info = super().step(
3138             action)
3139         self.time += self.dt
3140         if self.time % self.ball_spawn_interval < self.dt:
3141             ball_position = [np.random.uniform(low=-self.arena_size[0] /
3142                     2 + self.ball_radius, high=self.arena_size[0] / 2 - self.
3143                     ball_radius), np.random.uniform(low=-self.arena_size[1] /
3144                     2 + self.ball_radius, high=self.arena_size[1] / 2 - self.
3145                     ball_radius), 1.0]
3146             ball_velocity = [np.random.uniform(low=-1.0, high=1.0), np.
3147                 random.uniform(low=-1.0, high=1.0), 0.0]
3148             ball_velocity = self.ball_velocity * np.array(ball_velocity) /
3149                 np.linalg.norm(ball_velocity)
3150             ball_id = self.create_sphere(mass=1.0, radius=self.
3151                 ball_radius, position=ball_position, velocity=
3152                     ball_velocity, color=[1.0, 0.0, 0.0, 1.0])
3153             self.ball_ids.append(ball_id)
3154             for ball_id in self.ball_ids:
3155                 contact_points = self._p.getContactPoints(bodyA=self.robot.
3156                     robot_id, bodyB=ball_id)
3157                 if len(contact_points) > 0:
3158                     self.hit_by_ball = True
3159                     break
3160             return (observation, reward, terminated, truncated, info)
3161
3162     def get_task_rewards(self, action):
3163         survival = 1.0
3164         hit_penalty = -10.0 if self.hit_by_ball else 0.0
3165         return {'survival': survival, 'hit_penalty': hit_penalty}
3166
3167     def get_terminated(self, action):
3168         if self.hit_by_ball:
3169             return True
3170         if self.time >= self.survival_duration:
3171             return True
3172         return False
3173
3174     def get_success(self):
3175         return self.time >= self.survival_duration and (not self.
3176             hit_by_ball)

```

### Task 11

```

3172     import numpy as np
3173     from oped.envs.r2d2.base import R2D2Env
3174
3175     class Env(R2D2Env):
3176         """
3177             Jump from moving platform to moving platform to cross water and reach
3178             a target zone on a platform.
3179
3180             Description:
3181             - The environment consists of a start platform (3 m x 3 m x 0.5 m)
3182                 and an end platform (3 m x 3 m x 0.5 m) placed 20 meters apart.
3183             - The two platforms are connected by a series of 5 moving platforms
3184                 (2 m x 2 m x 0.5 m) placed 3 meters apart.
3185             - Each moving platform follows a sinusoidal motion along the y-axis
3186                 with an amplitude of 1 meter and a period of 3 seconds. The
3187                 motion of each platform is offset by 0.6 seconds from the
3188                 previous one.

```

```

3186
3187
3188
3189
3190
3191
3192
3193
3194
3195
3196
3197
3198
3199
3200
3201
3202
3203
3204
3205
3206
3207
3208
3209
3210
3211
3212
3213
3214
3215
3216
3217
3218
3219
3220
3221
3222
3223
3224
3225
3226
3227
3228
3229
3230
3231
3232
3233
3234
3235
3236
3237
3238
3239

```

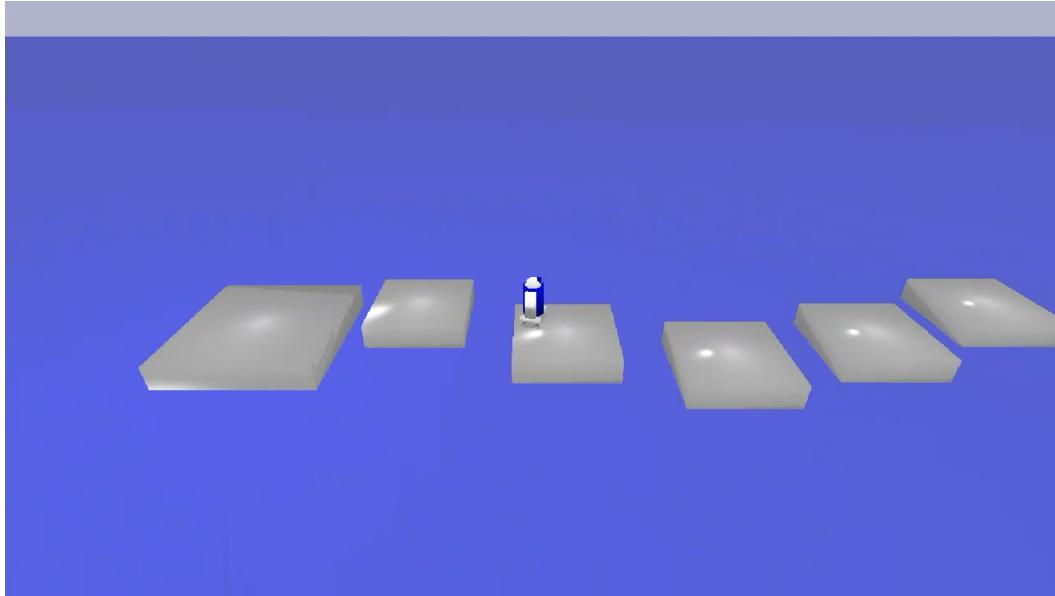


Figure 19: Task 11 of the OMNI-EPIC run presented in Section 5.

– The platforms are placed above a water surface, and the robot must jump from one moving platform to the next to reach the end platform.

**Success:**  
The task is successfully completed when the robot reaches the end platform.

**Rewards:**

- The robot receives a reward for each time step it remains on the platforms, encouraging steady progress.
- The robot is rewarded based on how much it reduces the distance to the end platform, incentivizing swift movement towards the goal.
- The robot is penalized for falling into the water.

**Termination:**  
The task terminates immediately if the robot falls into the water or reaches the end platform.  
\*\*\*

```

def __init__(self):
    super().__init__()
    self.water_size = [1000.0, 1000.0, 10.0]
    self.water_position = [0.0, 0.0, 0.0]
    self.water_id = self.create_box(mass=0.0, half_extents=[self.
        water_size[0] / 2, self.water_size[1] / 2, self.water_size[2]
        / 2], position=self.water_position, color=[0.0, 0.0, 1.0,
        0.5])
    self.platform_start_size = [3.0, 3.0, 0.5]
    self.platform_start_position = [0.0, 0.0, self.water_position[2]
        + self.water_size[2] / 2 + self.platform_start_size[2] / 2]
    self.platform_start_id = self.create_box(mass=0.0, half_extents=[

        self.platform_start_size[0] / 2, self.platform_start_size[1]
        / 2, self.platform_start_size[2] / 2], position=self.
        platform_start_position, color=[0.8, 0.8, 0.8, 1.0])
    self.platform_end_size = [3.0, 3.0, 0.5]

```

```

3240     self.platform_end_position = [self.platform_start_position[0] +
3241         20.0, self.platform_start_position[1], self.
3242         platform_start_position[2]]
3243     self.platform_end_id = self.create_box(mass=0.0, half_extents=[
3244         self.platform_end_size[0] / 2, self.platform_end_size[1] / 2,
3245         self.platform_end_size[2] / 2], position=self.
3246         platform_end_position, color=[0.8, 0.8, 0.8, 1.0])
3247     self.num_moving_platforms = 5
3248     self.moving_platform_size = [2.0, 2.0, 0.5]
3249     self.moving_platform_amplitude = 1.0
3250     self.moving_platform_period = 3.0
3251     self.moving_platform_phase_offset = 0.6
3252     self.moving_platform_ids = []
3253     for i in range(self.num_moving_platforms):
3254         moving_platform_position = [self.platform_start_position[0] +
3255             (i + 1) * 3.0, self.platform_start_position[1], self.
3256             platform_start_position[2]]
3257         moving_platform_id = self.create_box(mass=0.0, half_extents=[[
3258             self.moving_platform_size[0] / 2, self.moving_platform_size[1] /
3259             2, self.moving_platform_size[2] / 2], position=moving_platform_position, color=[0.8,
3260             0.8, 0.8, 1.0])
3261         self.moving_platform_ids.append(moving_platform_id)
3262
3263     def create_box(self, mass, half_extents, position, color):
3264         collision_shape_id = self._p.createCollisionShape(shapeType=self._p.GEOM_BOX, halfExtents=half_extents)
3265         visual_shape_id = self._p.createVisualShape(shapeType=self._p.GEOM_BOX, halfExtents=half_extents, rgbaColor=color)
3266         return self._p.createMultiBody(baseMass=mass,
3267             baseCollisionShapeIndex=collision_shape_id,
3268             baseVisualShapeIndex=visual_shape_id, basePosition=position)
3269
3270     def get_object_position(self, object_id):
3271         return np.asarray(self._p.getBasePositionAndOrientation(object_id)[0])
3272
3273     def get_distance_to_object(self, object_id):
3274         object_position = self.get_object_position(object_id)
3275         robot_position = self.robot.links['base'].position
3276         return np.linalg.norm(object_position[:2] - robot_position[:2])
3277
3278     def reset(self):
3279         observation = super().reset()
3280         self.time = 0.0
3281         self._p.resetBasePositionAndOrientation(self.robot.robot_id, [
3282             self.platform_start_position[0], self.platform_start_position[1],
3283             self.platform_start_position[2] + self.
3284             platform_start_size[2] / 2 + self.robot.links['base'].
3285             position_init[2]], self.robot.links['base'].orientation_init)
3286
3287     def step(self, action):
3288         self.distance_to_platform_end = self.get_distance_to_object(self.
3289             platform_end_id)
3290         observation, reward, terminated, truncated, info = super().step(
3291             action)
3292         self.time += self.dt
3293         for i, moving_platform_id in enumerate(self.moving_platform_ids):
3294             moving_platform_position = self.get_object_position(
3295                 moving_platform_id)
3296             new_moving_platform_position = [moving_platform_position[0],
3297                 self.platform_start_position[1] + self.
3298                 moving_platform_amplitude * np.sin(2 * np.pi * (self.time

```

```

3294         + i * self.moving_platform_phase_offset) / self.
3295         moving_platform_period), moving_platform_position[2])
3296         self._p.resetBasePositionAndOrientation(moving_platform_id,
3297             new_moving_platform_position, [0.0, 0.0, 0.0, 1.0])
3298     return (observation, reward, terminated, truncated, info)

3299     def get_task_rewards(self, action):
3300         new_distance_to_platform_end = self.get_distance_to_object(self.
3301             platform_end_id)
3302         on_platform = 1.0 if self.robot.links['base'].position[2] > self.
3303             platform_start_position[2] else -1.0
3304         reach_platform_end = (self.distance_to_platform_end -
3305             new_distance_to_platform_end) / self.dt
3306     return {'on_platform': on_platform, 'reach_platform_end':
3307         reach_platform_end}

3308     def get_terminated(self, action):
3309         is_in_water = self.robot.links['base'].position[2] < self.
3310             water_position[2] + self.water_size[2] / 2
3311         is_on_platform_end = self.get_distance_to_object(self.
3312             platform_end_id) < self.platform_end_size[0] / 2
3313         return is_in_water or is_on_platform_end

3314     def get_success(self):
3315         is_on_platform_end = self.get_distance_to_object(self.
3316             platform_end_id) < self.platform_end_size[0] / 2
3317         return is_on_platform_end

```

### Task 12

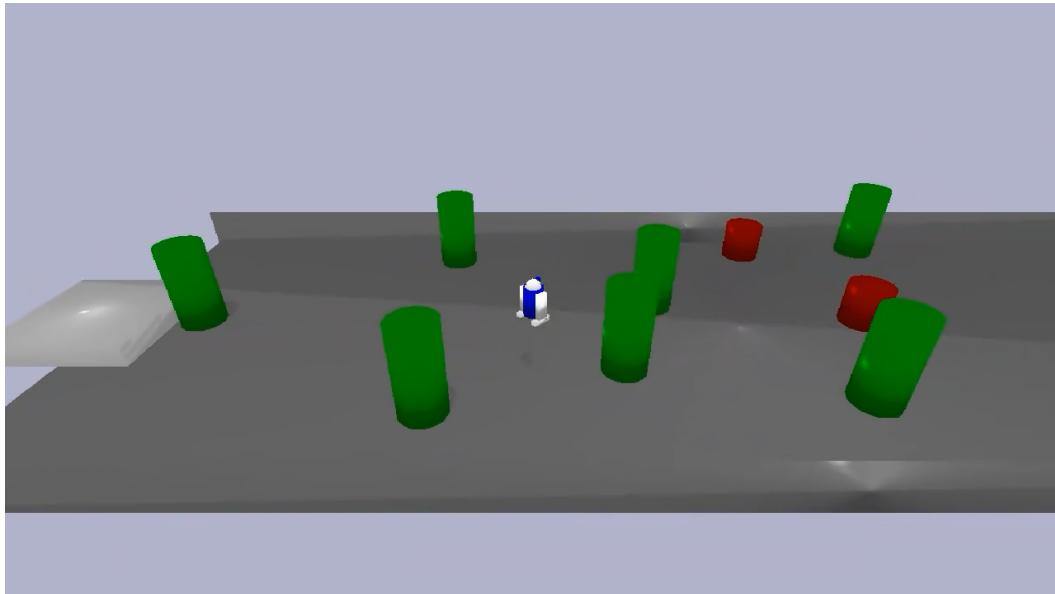


Figure 20: Task 12 of the OMNI-EPIC run presented in Section 5.

```

3341
3342     import numpy as np
3343     from oped.envs.r2d2.base import R2D2Env
3344
3345     class Env(R2D2Env):
3346         """
3347             Navigate a terrain with obstacles to reach a target zone.
3348
3349             Description:

```

```

3348     - The environment consists of a start platform (3 m x 3 m x 0.5 m)
3349         and an end platform (3 m x 3 m x 0.5 m) placed 30 meters apart.
3350     - The terrain between the platforms is filled with static obstacles
3351         such as rocks (cylinders) and trees (cylinders with a height of 3
3352         meters).
3353     - The obstacles are randomly placed with a minimum distance of 1
3354         meter between them, ensuring the robot cannot go around them
3355         easily.
3356     - The robot is initialized on the start platform, facing the end
3357         platform.

3358 Success:
3359 The task is successfully completed when the robot reaches the end
3360     platform.

3361 Rewards:
3362     - The robot receives a reward for each time step it remains on the
3363         platforms or the terrain, encouraging steady progress.
3364     - The robot is rewarded based on how much it reduces the distance to
3365         the end platform, incentivizing swift movement towards the goal.
3366     - The robot is penalized for collisions with obstacles.

3367 Termination:
3368 The task terminates immediately if the robot falls off the start
3369     platform, the terrain, or the end platform, or if the robot
3370     collides with an obstacle.
3371 """
3372
3373     def __init__(self):
3374         super().__init__()
3375         self.platform_size = [3.0, 3.0, 0.5]
3376         self.platform_start_position = [0.0, 0.0, 0.0]
3377         self.platform_end_position = [self.platform_start_position[0] +
3378             30.0, self.platform_start_position[1], self.
3379             platform_start_position[2]]
3380         self.platform_start_id = self.create_box(mass=0.0, half_extents=[

3381             self.platform_size[0] / 2, self.platform_size[1] / 2, self.
3382             platform_size[2] / 2], position=self.platform_start_position,
3383             color=[0.8, 0.8, 0.8, 1.0])
3384         self.platform_end_id = self.create_box(mass=0.0, half_extents=[

3385             self.platform_size[0] / 2, self.platform_size[1] / 2, self.
3386             platform_size[2] / 2], position=self.platform_end_position,
3387             color=[0.8, 0.8, 0.8, 1.0])
3388         self._p.changeDynamics(bodyUniqueId=self.platform_start_id,
3389             linkIndex=-1, lateralFriction=0.8, restitution=0.5)
3390         self._p.changeDynamics(bodyUniqueId=self.platform_end_id,
3391             linkIndex=-1, lateralFriction=0.8, restitution=0.5)
3392         self.terrain_size = [self.platform_end_position[0] - self.
3393             platform_start_position[0], 10.0, 0.1]
3394         self.terrain_position = [self.platform_start_position[0] + self.
3395             terrain_size[0] / 2, self.platform_start_position[1], self.
3396             platform_start_position[2] - self.platform_size[2] / 2 - self.
3397             terrain_size[2] / 2]
3398         self.terrain_id = self.create_box(mass=0.0, half_extents=[self.
3399             terrain_size[0] / 2, self.terrain_size[1] / 2, self.
3400             terrain_size[2] / 2], position=self.terrain_position, color
3401             =[0.5, 0.5, 0.5, 1.0])
3402         self._p.changeDynamics(bodyUniqueId=self.terrain_id, linkIndex
3403             =-1, lateralFriction=0.8, restitution=0.5)
3404         self.wall_size = [self.terrain_size[0], 0.1, 1.0]
3405         self.wall_left_position = [self.terrain_position[0], self.
3406             terrain_position[1] - self.terrain_size[1] / 2 - self.
3407             wall_size[1] / 2, self.terrain_position[2] + self.
3408             terrain_size[2] / 2 + self.wall_size[2] / 2]

```

```

3402     self.wall_right_position = [self.terrain_position[0], self.
3403         terrain_position[1] + self.terrain_size[1] / 2 + self.
3404         wall_size[1] / 2, self.terrain_position[2] + self.
3405         terrain_size[2] / 2 + self.wall_size[2] / 2]
3406     self.wall_left_id = self.create_box(mass=0.0, half_extents=[self.
3407         wall_size[0] / 2, self.wall_size[1] / 2, self.wall_size[2] /
3408         2], position=self.wall_left_position, color=[0.5, 0.5, 0.5,
3409         1.0])
3410     self.wall_right_id = self.create_box(mass=0.0, half_extents=[self.
3411         .wall_size[0] / 2, self.wall_size[1] / 2, self.wall_size[2] /
3412         2], position=self.wall_right_position, color=[0.5, 0.5, 0.5,
3413         1.0])
3414     self.num_obstacles = 10
3415     self.obstacle_radius = 0.5
3416     self.obstacle_height = 1.0
3417     self.tree_height = 3.0
3418     self.obstacle_ids = []
3419     for _ in range(self.num_obstacles):
3420         while True:
3421             obstacle_position = [np.random.uniform(self.
3422                 platform_start_position[0] + self.platform_size[0] /
3423                 2 + self.obstacle_radius, self.platform_end_position
3424                 [0] - self.platform_size[0] / 2 - self.
3425                 obstacle_radius), np.random.uniform(self.
3426                 terrain_position[1] - self.terrain_size[1] / 2 + self.
3427                 .obstacle_radius, self.terrain_position[1] + self.
3428                 terrain_size[1] / 2 - self.obstacle_radius), self.
3429                 terrain_position[2] + self.terrain_size[2] / 2 + self.
3430                 .obstacle_height / 2]
3431             if all((np.linalg.norm(np.array(obstacle_position[:2]) -
3432                 np.array(other_obstacle_position[:2])) > 2 * self.
3433                 obstacle_radius + 1.0 for other_obstacle_position in
3434                 [self.get_object_position(obstacle_id)[:2] for
3435                 obstacle_id in self.obstacle_ids])):
3436                 break
3437             if np.random.rand() < 0.5:
3438                 obstacle_id = self.create_cylinder(mass=0.0, radius=self.
3439                     obstacle_radius, height=self.obstacle_height,
3440                     position=obstacle_position, orientation=[0.0, 0.0,
3441                     0.0, 1.0], color=[0.5, 0.0, 0.0, 1.0])
3442             else:
3443                 obstacle_id = self.create_cylinder(mass=0.0, radius=self.
3444                     obstacle_radius, height=self.tree_height, position=
3445                     obstacle_position, orientation=[0.0, 0.0, 0.0, 1.0],
3446                     color=[0.0, 0.5, 0.0, 1.0])
3447             self.obstacle_ids.append(obstacle_id)
3448
3449     def create_box(self, mass, half_extents, position, color):
3450         collision_shape_id = self._p.createCollisionShape(shapeType=self.
3451             _p.GEOM_BOX, halfExtents=half_extents)
3452         visual_shape_id = self._p.createVisualShape(shapeType=self._p.
3453             GEOM_BOX, halfExtents=half_extents, rgbaColor=color)
3454         return self._p.createMultiBody(baseMass=mass,
3455             baseCollisionShapeIndex=collision_shape_id,
3456             baseVisualShapeIndex=visual_shape_id, basePosition=position)
3457
3458     def create_cylinder(self, mass, radius, height, position, orientation
3459     , color):
3460         collision_shape_id = self._p.createCollisionShape(shapeType=self.
3461             _p.GEOM_CYLINDER, radius=radius, height=height)
3462         visual_shape_id = self._p.createVisualShape(shapeType=self._p.
3463             GEOM_CYLINDER, radius=radius, length=height, rgbaColor=color)
3464         return self._p.createMultiBody(baseMass=mass,
3465             baseCollisionShapeIndex=collision_shape_id,

```

```

3456         baseVisualShapeIndex=visual_shape_id, basePosition=position,
3457         baseOrientation=orientation)
3458
3459     def get_object_position(self, object_id):
3460         return np.asarray(self._p.getBasePositionAndOrientation(object_id
3461                         )[0])
3462
3463     def get_distance_to_object(self, object_id):
3464         object_position = self.get_object_position(object_id)
3465         robot_position = self.robot.links['base'].position
3466         return np.linalg.norm(object_position[:2] - robot_position[:2])
3467
3468     def reset(self):
3469         observation = super().reset()
3470         self._p.resetBasePositionAndOrientation(self.robot.robot_id, [
3471             self.platform_start_position[0], self.platform_start_position
3472             [1], self.platform_start_position[2] + self.platform_size[2]
3473             / 2 + self.robot.links['base'].position_init[2]], self.robot.
3474             links['base'].orientation_init)
3475         return observation
3476
3477     def step(self, action):
3478         self.distance_to_platform_end = self.get_distance_to_object(self.
3479             platform_end_id)
3480         observation, reward, terminated, truncated, info = super().step(
3481             action)
3482         return (observation, reward, terminated, truncated, info)
3483
3484     def get_task_rewards(self, action):
3485         new_distance_to_platform_end = self.get_distance_to_object(self.
3486             platform_end_id)
3487         survival = 1.0
3488         reach_platform_end = (self.distance_to_platform_end -
3489             new_distance_to_platform_end) / self.dt
3490         collision = -1.0 if any((len(self._p.getContactPoints(bodyA=self.
3491             robot.robot_id, bodyB=obstacle_id)) > 0 for obstacle_id in
3492             self.obstacle_ids)) else 0.0
3493         return {'survival': survival, 'reach_platform_end':
3494             reach_platform_end, 'collision': collision}
3495
3496     def get_terminated(self, action):
3497         is_fall_off = self.robot.links['base'].position[2] < self.
3498             terrain_position[2]
3499         is_collision = any((len(self._p.getContactPoints(bodyA=self.robot
3500             .robot_id, bodyB=obstacle_id)) > 0 for obstacle_id in self.
3501             obstacle_ids))
3502         return is_fall_off or is_collision
3503
3504     def get_success(self):
3505         is_on_platform_end = self.get_distance_to_object(self.
3506             platform_end_id) < self.platform_size[0] / 2
3507         return is_on_platform_end

```

**Task 13**

```

3502 import numpy as np
3503 from oped.envs.r2d2.base import R2D2Env
3504
3505 class Env(R2D2Env):
3506     """
3507     Navigate a terrain with obstacles to reach a target zone.
3508
3509     Description:
3510     - The environment consists of a start platform (3 m x 3 m x 0.5 m)
3511       and an end platform (3 m x 3 m x 0.5 m) placed 30 meters apart.

```

A 3D rendering of a minimalist scene. A small, white rectangular robot with a blue cylindrical component on its back stands on a flat, grey surface. The surface is marked by several large, dark green cylinders of varying orientations. In the background, there's a plain, light blue wall.

Figure 21: Task 13 of the OMNI-EPIC run presented in Section 5.

```
3534     - The terrain between the platforms is filled with static obstacles  
3535         such as rocks (cylinders) and trees (cylinders with a height of 3  
3536             meters).  
3537     - The obstacles are placed in a grid pattern with enough space  
3538         between them to ensure a feasible path.  
3539     - The robot is initialized on the start platform, facing the end  
3540         platform.  
3541  
3542 Success:  
3543 The task is successfully completed when the robot reaches the end  
3544         platform.  
3545  
3546 Rewards:  
3547     - The robot receives a reward for each time step it remains on the  
3548         platforms or the terrain, encouraging steady progress.  
3549     - The robot is rewarded based on how much it reduces the distance to  
3550         the end platform, incentivizing swift movement towards the goal.  
3551     - The robot is penalized for collisions with obstacles.  
3552  
3553 Termination:  
3554 The task terminates if the robot falls off the start platform, the  
3555         terrain, or the end platform.  
3556 """  
3557  
3558 def __init__(self):  
3559     super().__init__()  
3560     self.platform_size = [3.0, 3.0, 0.5]  
3561     self.platform_start_position = [0.0, 0.0, 0.0]  
3562     self.platform_end_position = [self.platform_start_position[0] +  
3563         30.0, self.platform_start_position[1], self.  
3564         platform_start_position[2]]  
3565     self.platform_start_id = self.create_box(mass=0.0, half_extents=[  
3566         self.platform_size[0] / 2, self.platform_size[1] / 2, self.  
3567         platform_size[2] / 2], position=self.platform_start_position,  
3568         color=[0.8, 0.8, 0.8, 1.0])  
3569     self.platform_end_id = self.create_box(mass=0.0, half_extents=[  
3570         self.platform_size[0] / 2, self.platform_size[1] / 2, self.  
3571         platform_size[2] / 2], position=self.platform_end_position,
```

```

3564     platform_size[2] / 2], position=self.platform_end_position,
3565     color=[0.8, 0.8, 0.8, 1.0])
3566     self._p.changeDynamics(bodyUniqueId=self.platform_start_id,
3567         linkIndex=-1, lateralFriction=0.8, restitution=0.5)
3568     self._p.changeDynamics(bodyUniqueId=self.platform_end_id,
3569         linkIndex=-1, lateralFriction=0.8, restitution=0.5)
3570     self.terrain_size = [self.platform_end_position[0] - self.
3571         platform_start_position[0], 10.0, 0.1]
3572     self.terrain_position = [self.platform_start_position[0] + self.
3573         terrain_size[0] / 2, self.platform_start_position[1], self.
3574         platform_start_position[2] - self.platform_size[2] / 2 - self.
3575         terrain_size[2] / 2]
3576     self.terrain_id = self.create_box(mass=0.0, half_extents=[self.
3577         terrain_size[0] / 2, self.terrain_size[1] / 2, self.
3578         terrain_size[2] / 2], position=self.terrain_position, color
3579         =[0.5, 0.5, 0.5, 1.0])
3580     self._p.changeDynamics(bodyUniqueId=self.terrain_id, linkIndex
3581         =-1, lateralFriction=0.8, restitution=0.5)
3582     self.wall_size = [self.terrain_size[0], 0.1, 1.0]
3583     self.wall_left_position = [self.terrain_position[0], self.
3584         terrain_position[1] - self.terrain_size[1] / 2 - self.
3585         wall_size[1] / 2, self.terrain_position[2] + self.
3586         terrain_size[2] / 2 + self.wall_size[2] / 2]
3587     self.wall_right_position = [self.terrain_position[0], self.
3588         terrain_position[1] + self.terrain_size[1] / 2 + self.
3589         wall_size[1] / 2, self.terrain_position[2] + self.
3590         terrain_size[2] / 2 + self.wall_size[2] / 2]
3591     self.wall_left_id = self.create_box(mass=0.0, half_extents=[self.
3592         wall_size[0] / 2, self.wall_size[1] / 2, self.wall_size[2] /
3593         2], position=self.wall_left_position, color=[0.5, 0.5, 0.5,
3594         1.0])
3595     self.wall_right_id = self.create_box(mass=0.0, half_extents=[self.
3596         .wall_size[0] / 2, self.wall_size[1] / 2, self.wall_size[2] /
3597         2], position=self.wall_right_position, color=[0.5, 0.5, 0.5,
3598         1.0])
3599     self.num_obstacles = 10
3600     self.obstacle_radius = 0.5
3601     self.obstacle_height = 1.0
3602     self.tree_height = 3.0
3603     self.obstacle_ids = []
3604     self.place_obstacles()
3605
3606     def create_box(self, mass, half_extents, position, color):
3607         collision_shape_id = self._p.createCollisionShape(shapeType=self.
3608             _p.GEOM_BOX, halfExtents=half_extents)
3609         visual_shape_id = self._p.createVisualShape(shapeType=self._p.
3610             GEOM_BOX, halfExtents=half_extents, rgbaColor=color)
3611         return self._p.createMultiBody(baseMass=mass,
3612             baseCollisionShapeIndex=collision_shape_id,
3613             baseVisualShapeIndex=visual_shape_id, basePosition=position)
3614
3615     def create_cylinder(self, mass, radius, height, position, orientation
3616         , color):
3617         collision_shape_id = self._p.createCollisionShape(shapeType=self.
3618             _p.GEOM_CYLINDER, radius=radius, height=height)
3619         visual_shape_id = self._p.createVisualShape(shapeType=self._p.
3620             GEOM_CYLINDER, radius=radius, length=height, rgbaColor=color)
3621         return self._p.createMultiBody(baseMass=mass,
3622             baseCollisionShapeIndex=collision_shape_id,
3623             baseVisualShapeIndex=visual_shape_id, basePosition=position,
3624             baseOrientation=orientation)
3625
3626     def get_object_position(self, object_id):
3627         return np.asarray(self._p.getBasePositionAndOrientation(object_id
3628             )[0])

```

```

3618
3619     def get_distance_to_object(self, object_id):
3620         object_position = self.get_object_position(object_id)
3621         robot_position = self.robot.links['base'].position
3622         return np.linalg.norm(object_position[:2] - robot_position[:2])
3623
3624     def place_obstacles(self):
3625         grid_size = int(np.sqrt(self.num_obstacles))
3626         x_positions = np.linspace(self.platform_start_position[0] + self.
3627             platform_size[0] / 2 + self.obstacle_radius, self.
3628             platform_end_position[0] - self.platform_size[0] / 2 - self.
3629             obstacle_radius, grid_size)
3630         y_positions = np.linspace(self.terrain_position[1] - self.
3631             terrain_size[1] / 2 + self.obstacle_radius, self.
3632             terrain_position[1] + self.terrain_size[1] / 2 - self.
3633             obstacle_radius, grid_size)
3634         for x in x_positions:
3635             for y in y_positions:
3636                 if np.random.rand() < 0.5:
3637                     obstacle_id = self.create_cylinder(mass=0.0, radius=
3638                         self.obstacle_radius, height=self.obstacle_height
3639                         , position=[x, y, self.terrain_position[2] + self.
3640                         .terrain_size[2] / 2 + self.obstacle_height / 2],
3641                         orientation=[0.0, 0.0, 0.0, 1.0], color=[0.5,
3642                         0.5, 0.5, 1.0])
3643                 else:
3644                     obstacle_id = self.create_cylinder(mass=0.0, radius=
3645                         self.obstacle_radius, height=self.tree_height,
3646                         position=[x, y, self.terrain_position[2] + self.
3647                         .terrain_size[2] / 2 + self.tree_height / 2],
3648                         orientation=[0.0, 0.0, 0.0, 1.0], color=[0.0,
3649                         0.5, 0.0, 1.0])
3650             self.obstacle_ids.append(obstacle_id)
3651
3652     def reset(self):
3653         observation = super().reset()
3654         self._p.resetBasePositionAndOrientation(self.robot.robot_id, [
3655             self.platform_start_position[0], self.platform_start_position
3656             [1], self.platform_start_position[2] + self.platform_size[2]
3657             / 2 + self.robot.links['base'].position_init[2]], self.robot.
3658             links['base'].orientation_init)
3659         return observation
3660
3661     def step(self, action):
3662         self.distance_to_platform_end = self.get_distance_to_object(self.
3663             platform_end_id)
3664         observation, reward, terminated, truncated, info = super().step(
3665             action)
3666         return (observation, reward, terminated, truncated, info)
3667
3668     def get_task_rewards(self, action):
3669         new_distance_to_platform_end = self.get_distance_to_object(self.
3670             platform_end_id)
3671         survival = 1.0
3672         reach_platform_end = (self.distance_to_platform_end -
3673             new_distance_to_platform_end) / self.dt
3674         collision = -1.0 if any((len(self._p.getContactPoints(bodyA=self.
3675             robot.robot_id, bodyB=obstacle_id)) > 0 for obstacle_id in
3676             self.obstacle_ids)) else 0.0
3677         return {'survival': survival, 'reach_platform_end':
3678             reach_platform_end, 'collision': collision}
3679
3680     def get_terminated(self, action):
3681         is_fall_off = self.robot.links['base'].position[2] < self.
3682             terrain_position[2]

```

```

3672     is_collision = any((len(self._p.getContactPoints(bodyA=self.robot
3673         .robot_id, bodyB=obstacle_id)) > 0 for obstacle_id in self.
3674         obstacle_ids))
3675     return is_fall_off or is_collision
3676
3677     def get_success(self):
3678         is_on_platform_end = self.get_distance_to_object(self.
3679             platform_end_id) < self.platform_size[0] / 2
3680         return is_on_platform_end

```

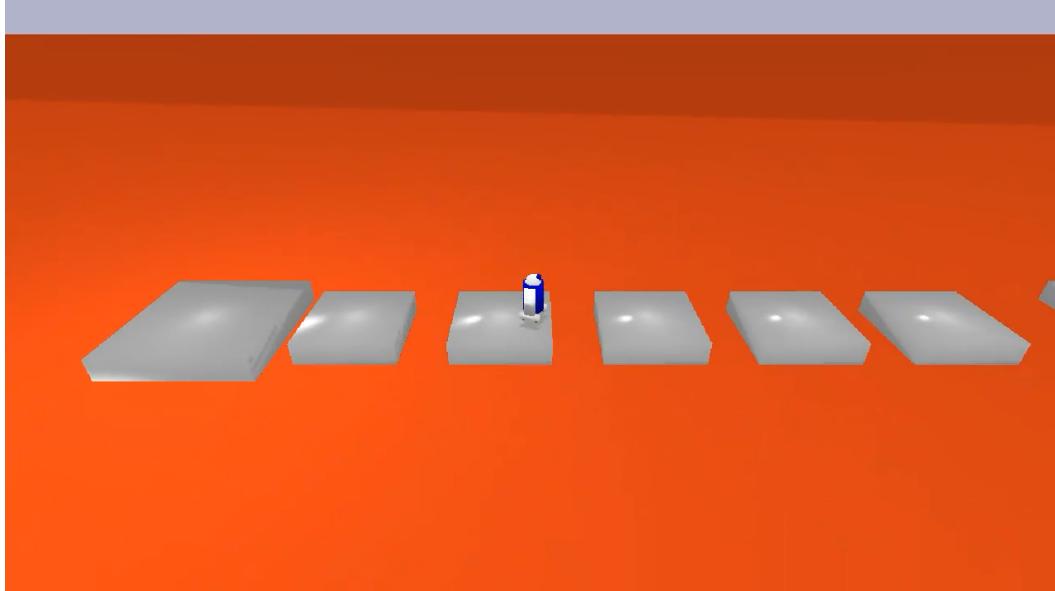
**Task 14**

Figure 22: Task 14 of the OMNI-EPIC run presented in Section 5.

```

3704
3705 import numpy as np
3706 from oped.envs.r2d2.base import R2D2Env
3707
3708 class Env(R2D2Env):
3709     """
3710         Jump from static platform to static platform to cross lava.
3711
3712     Description:
3713     - The environment consists of a start platform (3 m x 3 m x 0.5 m)
3714         and an end platform (3 m x 3 m x 0.5 m) placed 20 meters apart.
3715     - The two platforms are connected by a series of 5 static platforms
3716         (2 m x 2 m x 0.5 m) placed 3 meters apart.
3717     - The platforms are placed above a lava surface, and the robot must
3718         jump from one static platform to the next to reach the end
3719         platform.
3720
3721     Success:
3722     The task is successfully completed when the robot reaches the end
3723         platform.
3724
3725     Rewards:
3726     - The robot receives a reward for each time step it remains on the
3727         platforms, encouraging steady progress.
3728     - The robot is rewarded based on how much it reduces the distance to
3729         the end platform, incentivizing swift movement towards the goal.
3730     - The robot is penalized for falling into the lava.

```

```

3726
3727     Termination:
3728     The task terminates immediately if the robot falls into the lava or
3729     reaches the end platform.
3730     """
3731
3732     def __init__(self):
3733         super().__init__()
3734         self.lava_size = [1000.0, 1000.0, 10.0]
3735         self.lava_position = [0.0, 0.0, 0.0]
3736         self.lava_id = self.create_box(mass=0.0, half_extents=[self.
3737             lava_size[0] / 2, self.lava_size[1] / 2, self.lava_size[2] /
3738             2], position=self.lava_position, color=[1.0, 0.3, 0.1, 1.0])
3739         self.platform_start_size = [3.0, 3.0, 0.5]
3740         self.platform_start_position = [0.0, 0.0, self.lava_position[2] +
3741             self.lava_size[2] / 2 + self.platform_start_size[2] / 2]
3742         self.platform_start_id = self.create_box(mass=0.0, half_extents=[

3743             self.platform_start_size[0] / 2, self.platform_start_size[1] /
3744             2, self.platform_start_size[2] / 2], position=self.
3745             platform_start_position, color=[0.8, 0.8, 0.8, 1.0])
3746         self.platform_end_size = [3.0, 3.0, 0.5]
3747         self.platform_end_position = [self.platform_start_position[0] +
3748             20.0, self.platform_start_position[1], self.
3749             platform_start_position[2]]
3750         self.platform_end_id = self.create_box(mass=0.0, half_extents=[

3751             self.platform_end_size[0] / 2, self.platform_end_size[1] / 2,
3752             self.platform_end_size[2] / 2], position=self.
3753             platform_end_position, color=[0.8, 0.8, 0.8, 1.0])
3754         self.num_static_platforms = 5
3755         self.static_platform_size = [2.0, 2.0, 0.5]
3756         self.static_platform_ids = []
3757         for i in range(self.num_static_platforms):
3758             static_platform_position = [self.platform_start_position[0] +
3759                 (i + 1) * 3.0, self.platform_start_position[1], self.
3760                 platform_start_position[2]]
3761             static_platform_id = self.create_box(mass=0.0, half_extents=[

3762                 self.static_platform_size[0] / 2, self.
3763                 static_platform_size[1] / 2, self.static_platform_size[2] /
3764                 2], position=static_platform_position, color=[0.8,
3765                 0.8, 0.8, 1.0])
3766             self.static_platform_ids.append(static_platform_id)

3767     def create_box(self, mass, half_extents, position, color):
3768         collision_shape_id = self._p.createCollisionShape(shapeType=self.
3769             _p.GEOM_BOX, halfExtents=half_extents)
3770         visual_shape_id = self._p.createVisualShape(shapeType=self._p.
3771             GEOM_BOX, halfExtents=half_extents, rgbaColor=color)
3772         return self._p.createMultiBody(baseMass=mass,
3773             baseCollisionShapeIndex=collision_shape_id,
3774             baseVisualShapeIndex=visual_shape_id, basePosition=position)

3775     def get_object_position(self, object_id):
3776         return np.asarray(self._p.getBasePositionAndOrientation(object_id
3777             )[0])

3778     def get_distance_to_object(self, object_id):
3779         object_position = self.get_object_position(object_id)
3780         robot_position = self.robot.links['base'].position
3781         return np.linalg.norm(object_position[:2] - robot_position[:2])

3782     def reset(self):
3783         observation = super().reset()
3784         self._p.resetBasePositionAndOrientation(self.robot.robot_id, [
3785             self.platform_start_position[0], self.platform_start_position
3786             [1], self.platform_start_position[2] + self.

```

```

3780         platform_start_size[2] / 2 + self.robot.links['base'].  

3781         position_init[2]], self.robot.links['base'].orientation_init)  

3782     return observation  

3783  

3784     def step(self, action):  

3785         self.distance_to_platform_end = self.get_distance_to_object(self.  

3786             platform_end_id)  

3787         observation, reward, terminated, truncated, info = super().step(  

3788             action)  

3789         return (observation, reward, terminated, truncated, info)  

3790  

3791     def get_task_rewards(self, action):  

3792         new_distance_to_platform_end = self.get_distance_to_object(self.  

3793             platform_end_id)  

3794         on_platform = 1.0 if self.robot.links['base'].position[2] > self.  

3795             platform_start_position[2] else -1.0  

3796         reach_platform_end = (self.distance_to_platform_end -  

3797             new_distance_to_platform_end) / self.dt  

3798         return {'on_platform': on_platform, 'reach_platform_end':  

3799             reach_platform_end}  

3800  

3801     def get_terminated(self, action):  

3802         is_in_lava = self.robot.links['base'].position[2] < self.  

3803             lava_position[2] + self.lava_size[2] / 2  

3804         is_on_platform_end = self.get_distance_to_object(self.  

3805             platform_end_id) < self.platform_end_size[0] / 2  

3806         return is_in_lava or is_on_platform_end  

3807  

3808     Task 15

```

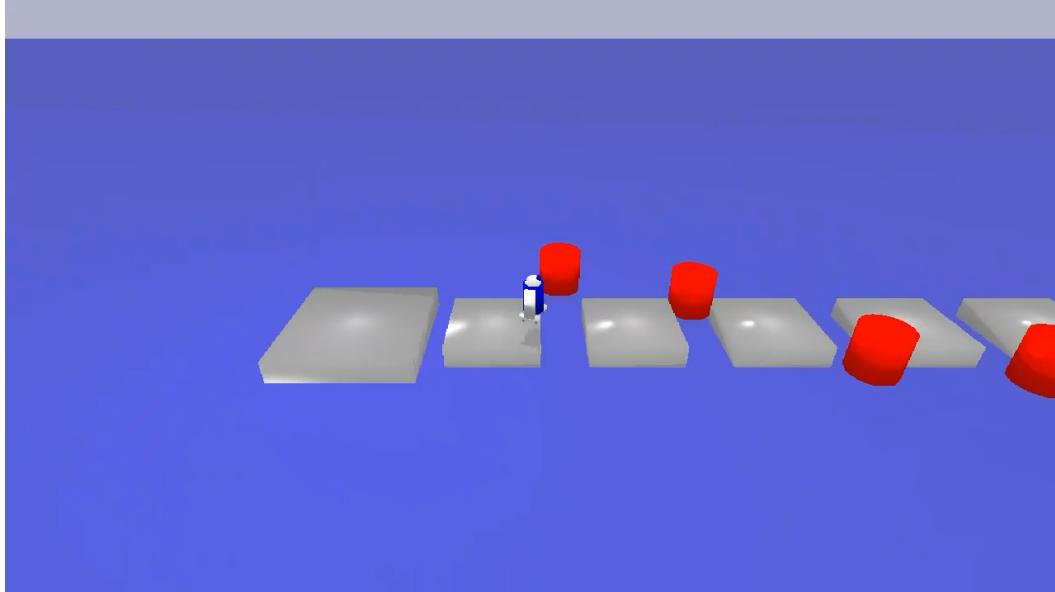


Figure 23: Task 15 of the OMNI-EPIC run presented in Section 5.

```

3829  

3830     import numpy as np  

3831     from oped.envs.r2d2.base import R2D2Env

```

```

3834 class Env(R2D2Env):
3835     """
3836     Jump from static platform to static platform while avoiding moving
3837     obstacles.
3838
3839     Description:
3840     - The environment consists of a start platform (3 m x 3 m x 0.5 m)
3841       and an end platform (3 m x 3 m x 0.5 m) placed 20 meters apart.
3842     - The two platforms are connected by a series of 5 static platforms
3843       (2 m x 2 m x 0.5 m) placed 3 meters apart.
3844     - Between these static platforms, moving obstacles (cylinders with a
3845       radius of 0.5 m and height of 1 m) move along the y-axis with a
3846       sinusoidal motion. The amplitude of the motion is 2 meters, and
3847       the period is 4 seconds. The motion of each obstacle is offset by
3848       0.8 seconds from the previous one.
3849     - The platforms and obstacles are placed above a water surface, and
3850       the robot must jump from one static platform to the next while
3851       avoiding the moving obstacles to reach the end platform.
3852
3853     Success:
3854     The task is successfully completed when the robot reaches the end
3855     platform.
3856
3857     Rewards:
3858     - The robot receives a reward for each time step it remains on the
3859       platforms, encouraging steady progress.
3860     - The robot is rewarded based on how much it reduces the distance to
3861       the end platform, incentivizing swift movement towards the goal.
3862     - The robot is penalized for collisions with moving obstacles.
3863
3864     Termination:
3865     The task terminates immediately if the robot falls into the water,
3866     collides with an obstacle, or reaches the end platform.
3867     """
3868
3869     def __init__(self):
3870         super().__init__()
3871         self.water_size = [1000.0, 1000.0, 10.0]
3872         self.water_position = [0.0, 0.0, 0.0]
3873         self.water_id = self.create_box(mass=0.0, half_extents=[self.
3874             water_size[0] / 2, self.water_size[1] / 2, self.water_size[2]
3875             / 2], position=self.water_position, color=[0.0, 0.0, 1.0,
3876             0.5])
3877         self.platform_start_size = [3.0, 3.0, 0.5]
3878         self.platform_start_position = [0.0, 0.0, self.water_position[2]
3879             + self.water_size[2] / 2 + self.platform_start_size[2] / 2]
3880         self.platform_start_id = self.create_box(mass=0.0, half_extents=[

3881             self.platform_start_size[0] / 2, self.platform_start_size[1]
3882             / 2, self.platform_start_size[2] / 2], position=self.
3883             platform_start_position, color=[0.8, 0.8, 0.8, 1.0])
3884         self.platform_end_size = [3.0, 3.0, 0.5]
3885         self.platform_end_position = [self.platform_start_position[0] +
3886             20.0, self.platform_start_position[1], self.
3887             platform_start_position[2]]
3888         self.platform_end_id = self.create_box(mass=0.0, half_extents=[

3889             self.platform_end_size[0] / 2, self.platform_end_size[1] / 2,
3890             self.platform_end_size[2] / 2], position=self.
3891             platform_end_position, color=[0.8, 0.8, 0.8, 1.0])
3892         self.num_static_platforms = 5
3893         self.static_platform_size = [2.0, 2.0, 0.5]
3894         self.static_platform_ids = []
3895         for i in range(self.num_static_platforms):
3896             static_platform_position = [self.platform_start_position[0] +
3897                 (i + 1) * 3.0, self.platform_start_position[1], self.
3898                 platform_start_position[2]]

```

```

3888     static_platform_id = self.create_box(mass=0.0, half_extents=[  

3889         self.static_platform_size[0] / 2, self.  

3890         static_platform_size[1] / 2, self.static_platform_size[2]  

3891         / 2], position=static_platform_position, color=[0.8,  

3892         0.8, 0.8, 1.0])  

3893     self.static_platform_ids.append(static_platform_id)  

3894     self.num_moving_obstacles = 4  

3895     self.moving_obstacle_radius = 0.5  

3896     self.moving_obstacle_height = 1.0  

3897     self.moving_obstacle_amplitude = 2.0  

3898     self.moving_obstacle_period = 4.0  

3899     self.moving_obstacle_phase_offset = 0.8  

3900     self.moving_obstacle_ids = []  

3901     for i in range(self.num_moving_obstacles):  

3902         moving_obstacle_position = [self.platform_start_position[0] +  

3903             (i + 1) * 3.0 + 1.5, self.platform_start_position[1],  

3904             self.platform_start_position[2] + self.  

3905             moving_obstacle_height / 2]  

3906         moving_obstacle_id = self.create_cylinder(mass=0.0, radius=  

3907             self.moving_obstacle_radius, height=self.  

3908             moving_obstacle_height, position=moving_obstacle_position  

3909             , color=[1.0, 0.0, 0.0, 1.0])  

3910         self.moving_obstacle_ids.append(moving_obstacle_id)  

3911  

3912     def create_box(self, mass, half_extents, position, color):  

3913         collision_shape_id = self._p.createCollisionShape(shapeType=self.  

3914             _p.GEOM_BOX, halfExtents=half_extents)  

3915         visual_shape_id = self._p.createVisualShape(shapeType=self._p.  

3916             GEOM_BOX, halfExtents=half_extents, rgbaColor=color)  

3917         return self._p.createMultiBody(baseMass=mass,  

3918             baseCollisionShapeIndex=collision_shape_id,  

3919             baseVisualShapeIndex=visual_shape_id, basePosition=position)  

3920  

3921     def create_cylinder(self, mass, radius, height, position, color):  

3922         collision_shape_id = self._p.createCollisionShape(shapeType=self.  

3923             _p.GEOM_CYLINDER, radius=radius, height=height)  

3924         visual_shape_id = self._p.createVisualShape(shapeType=self._p.  

3925             GEOM_CYLINDER, radius=radius, length=height, rgbaColor=color)  

3926         return self._p.createMultiBody(baseMass=mass,  

3927             baseCollisionShapeIndex=collision_shape_id,  

3928             baseVisualShapeIndex=visual_shape_id, basePosition=position)  

3929  

3930     def get_object_position(self, object_id):  

3931         return np.asarray(self._p.getBasePositionAndOrientation(object_id  

3932             )[0])  

3933  

3934     def get_distance_to_object(self, object_id):  

3935         object_position = self.get_object_position(object_id)  

3936         robot_position = self.robot.links['base'].position  

3937         return np.linalg.norm(object_position[:2] - robot_position[:2])  

3938  

3939     def reset(self):  

3940         observation = super().reset()  

3941         self.time = 0.0  

3942         self._p.resetBasePositionAndOrientation(self.robot.robot_id, [  

3943             self.platform_start_position[0], self.platform_start_position  

3944             [1], self.platform_start_position[2] + self.  

3945             platform_start_size[2] / 2 + self.robot.links['base'].  

3946             position_init[2]], self.robot.links['base'].orientation_init)  

3947         return observation  

3948  

3949     def step(self, action):  

3950         self.distance_to_platform_end = self.get_distance_to_object(self.  

3951             platform_end_id)

```

```

3942     observation, reward, terminated, truncated, info = super().step(
3943         action)
3944     self.time += self.dt
3945     for i, moving_obstacle_id in enumerate(self.moving_obstacle_ids):
3946         moving_obstacle_position = self.get_object_position(
3947             moving_obstacle_id)
3948         new_moving_obstacle_position = [moving_obstacle_position[0],
3949             self.platform_start_position[1] + self.
3950             moving_obstacle_amplitude * np.sin(2 * np.pi * (self.time
3951                 + i * self.moving_obstacle_phase_offset) / self.
3952                 moving_obstacle_period), moving_obstacle_position[2]]
3953         self._p.resetBasePositionAndOrientation(moving_obstacle_id,
3954             new_moving_obstacle_position, [0.0, 0.0, 0.0, 1.0])
3955     return (observation, reward, terminated, truncated, info)
3956
3957     def get_task_rewards(self, action):
3958         new_distance_to_platform_end = self.get_distance_to_object(self.
3959             platform_end_id)
3960         on_platform = 1.0 if self.robot.links['base'].position[2] > self.
3961             platform_start_position[2] else -1.0
3962         reach_platform_end = (self.distance_to_platform_end -
3963             new_distance_to_platform_end) / self.dt
3964         collision_with_obstacle = -1.0 if len(self._p.getContactPoints(
3965             bodyA=self.robot.robot_id, bodyB=self.moving_obstacle_ids[0]))
3966                 > 0 or len(self._p.getContactPoints(bodyA=self.robot.
3967                     robot_id, bodyB=self.moving_obstacle_ids[1])) > 0 or len(self.
3968                     _p.getContactPoints(bodyA=self.robot.robot_id, bodyB=self.
3969                         moving_obstacle_ids[2])) > 0 or (len(self._p.getContactPoints(
3970                             bodyA=self.robot.robot_id, bodyB=self.moving_obstacle_ids
3971                             [3])) > 0) else 0.0
3972         return {'on_platform': on_platform, 'reach_platform_end':
3973             reach_platform_end, 'collision_with_obstacle':
3974                 collision_with_obstacle}
3975
3976     def get_terminated(self, action):
3977         is_in_water = self.robot.links['base'].position[2] < self.
3978             water_position[2] + self.water_size[2] / 2
3979         is_on_platform_end = self.get_distance_to_object(self.
3980             platform_end_id) < self.platform_end_size[0] / 2
3981         collision_with_obstacle = len(self._p.getContactPoints(bodyA=self.
3982             .robot.robot_id, bodyB=self.moving_obstacle_ids[0])) > 0 or
3983             len(self._p.getContactPoints(bodyA=self.robot.robot_id, bodyB
3984                 =self.moving_obstacle_ids[1])) > 0 or len(self._p.
3985                 getContactPoints(bodyA=self.robot.robot_id, bodyB=self.
3986                     moving_obstacle_ids[2])) > 0 or (len(self._p.getContactPoints(
3987                         bodyA=self.robot.robot_id, bodyB=self.moving_obstacle_ids
3988                         [3])) > 0)
3989         return is_in_water or is_on_platform_end or
3990             collision_with_obstacle
3991
3992     def get_success(self):
3993         is_on_platform_end = self.get_distance_to_object(self.
3994             platform_end_id) < self.platform_end_size[0] / 2
3995         return is_on_platform_end

```

**Task 16**

```

import numpy as np
from oped.envs.r2d2.base import R2D2Env

class Env(R2D2Env):
    """
    Task: Jump over rolling logs on a bridge
    Description:

```

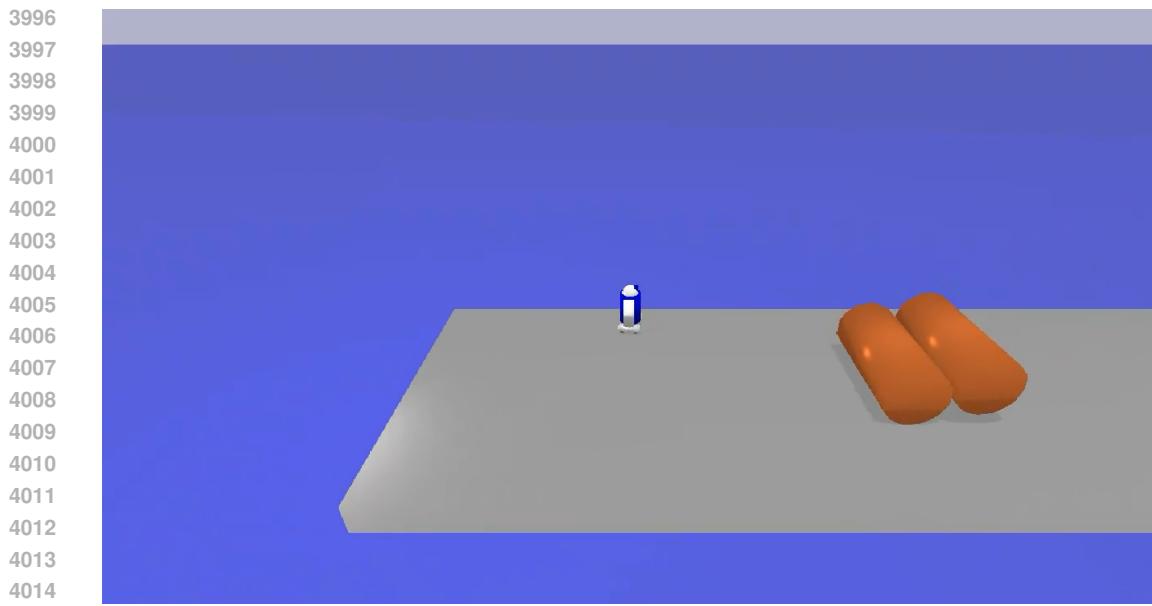


Figure 24: Task 16 of the OMNI-EPIC run presented in Section 5.

4019     – The environment consists of a wide static bridge that is 50 meters  
 4020       long and 5 meters wide, elevated 10 meters above a water surface.  
 4021     – The robot starts at the beginning of the bridge, facing the  
       positive x-axis.  
 4022     – Logs (cylinders) with a radius of 0.5 meters are spawned 2 meters  
 4023       above the bridge and 10 meters ahead of the robot, rolling  
       towards the robot with an initial velocity of 5 m/s along the  
       negative x-axis.  
 4024     – The robot must move forward on the bridge while jumping over the  
 4025       rolling logs to reach the end of the bridge.  
 4026  
 4027  
 4028     **Success:**  
 4029       The task is successfully completed when the robot reaches the end of  
 4030       the bridge.  
 4031  
 4032     **Rewards:**  
 4033       – The robot receives a survival reward for each time step it remains  
       on the bridge, encouraging steady progress.  
 4034       – The robot is rewarded based on how much it reduces the distance to  
       the end of the bridge, incentivizing swift movement towards the  
       goal.  
 4035       – The robot is penalized for colliding with the logs.  
 4036  
 4037  
 4038     **Termination:**  
 4039       The task terminates immediately if the robot falls off the bridge,  
 4040       collides with a log, or reaches the end of the bridge.  
 4041       \*\*\*  
 4042  
 4043     def \_\_init\_\_(self):  
 4044       super().\_\_init\_\_()  
 4045       self.water\_size = [1000.0, 1000.0, 10.0]  
 4046       self.water\_position = [0.0, 0.0, 0.0]  
 4047       self.water\_id = self.create\_box(mass=0.0, half\_extents=[self.  
           water\_size[0] / 2, self.water\_size[1] / 2, self.water\_size[2]  
           / 2], position=self.water\_position, color=[0.0, 0.0, 1.0,  
           0.5])  
 4048       self.bridge\_length = 50.0  
 4049       self.bridge\_width = 5.0

```

4050     self.bridge_height = 0.5
4051     self.bridge_position = [self.water_position[0] + self.
4052         bridge_length / 2, self.water_position[1], self.
4053         water_position[2] + self.water_size[2] / 2 + self.
4054         bridge_height / 2]
4055     self.bridge_id = self.create_box(mass=0.0, half_extents=[self.
4056         bridge_length / 2, self.bridge_width / 2, self.bridge_height
4057         / 2], position=self.bridge_position, color=[0.8, 0.8, 0.8,
4058         1.0])
4059     self.log_radius = 0.5
4060     self.log_height = 2.0
4061     self.log_spawn_distance = 10.0
4062     self.log_spawn_height = 2.0
4063     self.log_velocity = -5.0
4064     self.log_ids = []
4065     self.robot_position_init = [self.bridge_position[0] - self.
4066         bridge_length / 2 + 1.0, self.bridge_position[1], self.
4067         bridge_position[2] + self.bridge_height / 2 + self.robot.
4068         links['base'].position_init[2]]
4069     self.robot_orientation_init = self._p.getQuaternionFromEuler
4070         ([0.0, 0.0, 0.0])
4071
4072     def create_box(self, mass, half_extents, position, color):
4073         collision_shape_id = self._p.createCollisionShape(shapeType=self.
4074             _p.GEOM_BOX, halfExtents=half_extents)
4075         visual_shape_id = self._p.createVisualShape(shapeType=self._p.
4076             GEOM_BOX, halfExtents=half_extents, rgbaColor=color)
4077         return self._p.createMultiBody(baseMass=mass,
4078             baseCollisionShapeIndex=collision_shape_id,
4079             baseVisualShapeIndex=visual_shape_id, basePosition=position)
4080
4081     def create_cylinder(self, mass, radius, height, position, orientation
4082         , color):
4083         collision_shape_id = self._p.createCollisionShape(shapeType=self.
4084             _p.GEOM_CYLINDER, radius=radius, height=height)
4085         visual_shape_id = self._p.createVisualShape(shapeType=self._p.
4086             GEOM_CYLINDER, radius=radius, length=height, rgbaColor=color)
4087         return self._p.createMultiBody(baseMass=mass,
4088             baseCollisionShapeIndex=collision_shape_id,
4089             baseVisualShapeIndex=visual_shape_id, basePosition=position,
4090             baseOrientation=orientation)
4091
4092     def get_object_position(self, object_id):
4093         return np.asarray(self._p.getBasePositionAndOrientation(object_id
4094             )[0])
4095
4096     def get_distance_to_object(self, object_id):
4097         object_position = self.get_object_position(object_id)
4098         robot_position = self.robot.links['base'].position
4099         return np.linalg.norm(object_position[:2] - robot_position[:2])
4100
4101     def reset(self):
4102         observation = super().reset()
4103         self._p.resetBasePositionAndOrientation(self.robot.robot_id, self.
4104             .robot_position_init, self.robot_orientation_init)
4105         for log_id in self.log_ids:
4106             self._p.removeBody(log_id)
4107         self.log_ids = []
4108         return observation
4109
4110     def step(self, action):
4111         self.distance_to_bridge_end = self.bridge_length - (self.robot.
4112             links['base'].position[0] - (self.bridge_position[0] - self.
4113                 bridge_length / 2))

```

```

4104     observation, reward, terminated, truncated, info = super().step(
4105         action)
4106     if np.random.rand() < 0.05:
4107         log_position = [self.robot.links['base'].position[0] + self.
4108             log_spawn_distance, self.bridge_position[1], self.
4109             bridge_position[2] + self.bridge_height / 2 + self.
4110             log_spawn_height]
4111     log_orientation = self._p.getQuaternionFromEuler([np.pi / 2,
4112         0.0, 0.0])
4113     log_id = self.create_cylinder(mass=10.0, radius=self.
4114         log_radius, height=self.log_height, position=log_position
4115         , orientation=log_orientation, color=[0.8, 0.4, 0.2,
4116         1.0])
4117     self._p.resetBaseVelocity(log_id, linearVelocity=[self.
4118         log_velocity, 0.0, 0.0])
4119     self.log_ids.append(log_id)
4120     for log_id in self.log_ids:
4121         log_position = self.get_object_position(log_id)
4122         if log_position[0] < self.bridge_position[0] - self.
4123             bridge_length / 2:
4124             self._p.removeBody(log_id)
4125             self.log_ids.remove(log_id)
4126     return (observation, reward, terminated, truncated, info)
4127
4128     def get_task_rewards(self, action):
4129         new_distance_to_bridge_end = self.bridge_length - (self.robot.
4130             links['base'].position[0] - (self.bridge_position[0] - self.
4131                 bridge_length / 2))
4132         survival = 1.0 if self.robot.links['base'].position[2] > self.
4133             bridge_position[2] else -1.0
4134         reach_bridge_end = (self.distance_to_bridge_end -
4135             new_distance_to_bridge_end) / self.dt
4136         collision_with_log = -1.0 if any((len(self._p.getContactPoints(
4137             bodyA=self.robot.robot_id, bodyB=log_id)) > 0 for log_id in
4138                 self.log_ids)) else 0.0
4139         return {'survival': survival, 'reach_bridge_end':
4140             reach_bridge_end, 'collision_with_log': collision_with_log}
4141
4142     def get_terminated(self, action):
4143         is_off_bridge = self.robot.links['base'].position[2] < self.
4144             bridge_position[2]
4145         is_at_bridge_end = self.robot.links['base'].position[0] > self.
4146             bridge_position[0] + self.bridge_length / 2
4147         collision_with_log = any((len(self._p.getContactPoints(bodyA=self.
4148             .robot.robot_id, bodyB=log_id)) > 0 for log_id in self.
4149                 log_ids))
4150         return is_off_bridge or is_at_bridge_end or collision_with_log
4151
4152     def get_success(self):
4153         is_at_bridge_end = self.robot.links['base'].position[0] > self.
4154             bridge_position[0] + self.bridge_length / 2
4155         return is_at_bridge_end

```

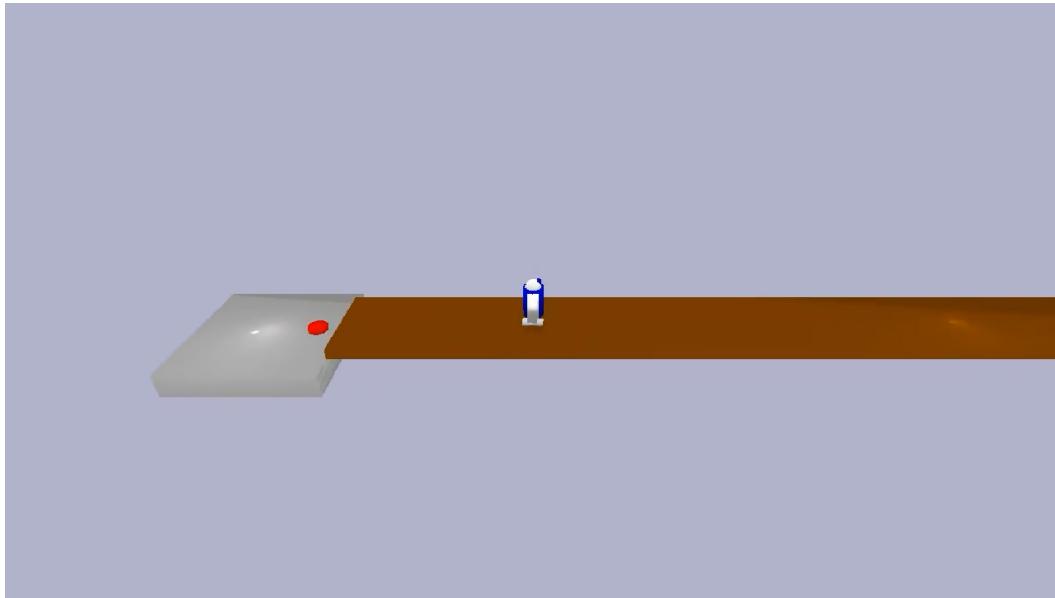
**Task 17**

```

4150     import numpy as np
4151     from oped.envs.r2d2.base import R2D2Env
4152
4153     class Env(R2D2Env):
4154         """
4155             Touch a lever or a button to activate a drawbridge. The robot is on a
4156             start platform and needs to reach a target platform 25 m away. A
4157             25-meter long drawbridge is on the end platform in an open (
4158             upright) position. A button or lever can be activated to lower
4159             the drawbridge to connect the two platforms.

```

4158  
4159  
4160  
4161  
4162  
4163  
4164  
4165  
4166  
4167  
4168  
4169  
4170  
4171  
4172  
4173  
4174  
4175  
4176  
4177



4178  
4179  
4180  
4181

Figure 25: Task 17 of the OMNI-EPIC run presented in Section 5.

4182  
4183  
4184  
4185  
4186  
4187  
4188  
4189

**Description:**

- The environment consists of a start platform ( $3 \text{ m} \times 3 \text{ m} \times 0.5 \text{ m}$ ) and an end platform ( $3 \text{ m} \times 3 \text{ m} \times 0.5 \text{ m}$ ) placed 25 meters apart.
- A drawbridge (25 m long and 2 m wide) is positioned at the edge of the end platform in an upright position.
- A button or lever is placed on the start platform.
- The robot must touch the button or lever to lower the drawbridge, allowing it to cross to the end platform.

4190  
4191  
4192

**Success:**

The task is successfully completed when the robot reaches the end platform.

4193  
4194  
4195  
4196  
4197  
4198

**Rewards:**

- The robot receives a reward for each time step it remains on the platforms, encouraging steady progress.
- The robot is rewarded for touching the button or lever.
- The robot is rewarded based on how much it reduces the distance to the end platform, incentivizing swift movement towards the goal.

4199  
4200  
4201  
4202

**Termination:**

The task terminates immediately if the robot falls off the platforms or reaches the end platform.

\*\*\*\*

4203  
4204  
4205  
4206  
4207  
4208  
4209  
4210  
4211

```
def __init__(self):
    super().__init__()
    self.platform_size = [3.0, 3.0, 0.5]
    self.platform_start_position = [0.0, 0.0, 0.0]
    self.platform_end_position = [self.platform_start_position[0] +
        25.0, self.platform_start_position[1], self.
        platform_start_position[2]]
    self.platform_start_id = self.create_box(mass=0.0, half_extents=[

        self.platform_size[0] / 2, self.platform_size[1] / 2, self.
        platform_size[2] / 2], position=self.platform_start_position,
        color=[0.8, 0.8, 0.8, 1.0])
```

```

4212     self.platform_end_id = self.create_box(mass=0.0, half_extents=[
4213         self.platform_size[0] / 2, self.platform_size[1] / 2, self.
4214         platform_size[2] / 2], position=self.platform_end_position,
4215         color=[0.8, 0.8, 0.8, 1.0])
4216     self.drawbridge_length = 25.0
4217     self.drawbridge_width = 2.0
4218     self.drawbridge_thickness = 0.2
4219     self.drawbridge_position_lowered = [self.platform_start_position
4220         [0] + self.platform_size[0] / 2 + self.drawbridge_length / 2,
4221         self.platform_start_position[1], self.
4222         platform_start_position[2] + self.platform_size[2] / 2 + self
4223         .drawbridge_thickness / 2]
4224     self.drawbridge_position_raised = [self.platform_end_position[0]
4225         - self.platform_size[0] / 2, self.platform_end_position[1],
4226         self.platform_end_position[2] + self.platform_size[2] / 2 +
4227         self.drawbridge_length / 2]
4228     self.drawbridge_id = self.create_box(mass=0.0, half_extents=[self
4229         .drawbridge_length / 2, self.drawbridge_width / 2, self.
4230         drawbridge_thickness / 2], position=self.
4231         drawbridge_position_raised, color=[0.6, 0.3, 0.0, 1.0])
4232     self.button_radius = 0.2
4233     self.button_height = 0.1
4234     self.button_position = [self.platform_start_position[0] + self.
4235         platform_size[0] / 2 - 0.5, self.platform_start_position[1],
4236         self.platform_start_position[2] + self.platform_size[2] / 2 +
4237         self.button_height / 2]
4238     self.button_id = self.create_cylinder(mass=0.0, radius=self.
4239         button_radius, height=self.button_height, position=self.
4240         button_position, color=[1.0, 0.0, 0.0, 1.0])
4241     self.drawbridge_activated = False
4242
4243     def create_box(self, mass, half_extents, position, color):
4244         collision_shape_id = self._p.createCollisionShape(shapeType=self.
4245             _p.GEOM_BOX, halfExtents=half_extents)
4246         visual_shape_id = self._p.createVisualShape(shapeType=self._p.
4247             GEOM_BOX, halfExtents=half_extents, rgbaColor=color)
4248         return self._p.createMultiBody(baseMass=mass,
4249             baseCollisionShapeIndex=collision_shape_id,
4250             baseVisualShapeIndex=visual_shape_id, basePosition=position)
4251
4252     def create_cylinder(self, mass, radius, height, position, color):
4253         collision_shape_id = self._p.createCollisionShape(shapeType=self.
4254             _p.GEOM_CYLINDER, radius=radius, height=height)
4255         visual_shape_id = self._p.createVisualShape(shapeType=self._p.
4256             GEOM_CYLINDER, radius=radius, length=height, rgbaColor=color)
4257         return self._p.createMultiBody(baseMass=mass,
4258             baseCollisionShapeIndex=collision_shape_id,
4259             baseVisualShapeIndex=visual_shape_id, basePosition=position)
4260
4261     def get_object_position(self, object_id):
4262         return np.asarray(self._p.getBasePositionAndOrientation(object_id
4263             )[0])
4264
4265     def get_distance_to_object(self, object_id):
4266         object_position = self.get_object_position(object_id)
4267         robot_position = self.robot.links['base'].position
4268         return np.linalg.norm(object_position[:2] - robot_position[:2])
4269
4270     def reset(self):
4271         observation = super().reset()
4272         self._p.resetBasePositionAndOrientation(self.robot.robot_id, [
4273             self.platform_start_position[0], self.platform_start_position
4274             [1], self.platform_start_position[2] + self.platform_size[2]
4275             / 2 + self.robot.links['base'].position_init[2]], self.robot.
4276             links['base'].orientation_init)

```

```

4266     self.drawbridge_activated = False
4267     self._p.resetBasePositionAndOrientation(self.drawbridge_id, self.
4268         drawbridge_position_raised, [0.0, 0.0, 0.0, 1.0])
4269     return observation
4270
4271     def step(self, action):
4272         self.distance_to_platform_end = self.get_distance_to_object(self.
4273             platform_end_id)
4274         observation, reward, terminated, truncated, info = super().step(
4275             action)
4276         if not self.drawbridge_activated and len(self._p.getContactPoints
4277             (bodyA=self.robot.robot_id, bodyB=self.button_id)) > 0:
4278             self.drawbridge_activated = True
4279             self._p.resetBasePositionAndOrientation(self.drawbridge_id,
4280                 self.drawbridge_position_lowered, [0.0, 0.0, 0.0, 1.0])
4281         return (observation, reward, terminated, truncated, info)
4282
4283     def get_task_rewards(self, action):
4284         new_distance_to_platform_end = self.get_distance_to_object(self.
4285             platform_end_id)
4286         on_platforms = 1.0 if self.robot.links['base'].position[2] > self.
4287             .platform_start_position[2] + self.platform_size[2] / 2 else
4288             -1.0
4289         activate_drawbridge = 10.0 if not self.drawbridge_activated and
4290             len(self._p.getContactPoints(bodyA=self.robot.robot_id, bodyB
4291             =self.button_id)) > 0 else 0.0
4292         reach_platform_end = (self.distance_to_platform_end -
4293             new_distance_to_platform_end) / self.dt
4294         return {'on_platforms': on_platforms, 'activate_drawbridge':
4295             activate_drawbridge, 'reach_platform_end': reach_platform_end
4296             }
4297
4298     def get_terminated(self, action):
4299         is_off_platforms = self.robot.links['base'].position[2] < self.
4300             platform_start_position[2]
4301         is_on_platform_end = self.get_distance_to_object(self.
4302             platform_end_id) < self.platform_size[0] / 2
4303         return is_off_platforms or is_on_platform_end
4304
4305     def get_success(self):
4306         is_on_platform_end = self.get_distance_to_object(self.
4307             platform_end_id) < self.platform_size[0] / 2
4308         return is_on_platform_end

```

### Task 18

```

4305 import numpy as np
4306 from oped.envs.r2d2.base import R2D2Env
4307
4308 class Env(R2D2Env):
4309     """
4310     Task: Push a cube to a target zone on a static platform
4311
4312     Description:
4313     - The environment consists of a large static platform (50 m x 50 m).
4314     - A cube (2 meters in size and 5 kg in mass) is placed at a random
4315         location on the platform.
4316     - A target zone (3 meters in radius) is also placed at a random
4317         location on the platform. The collision for the target zone is
4318         set to False.
4319     - The robot is initialized at a fixed position on the platform.
4320
4321     The task of the robot is to push the cube to the target zone as
4322         quickly as possible.

```

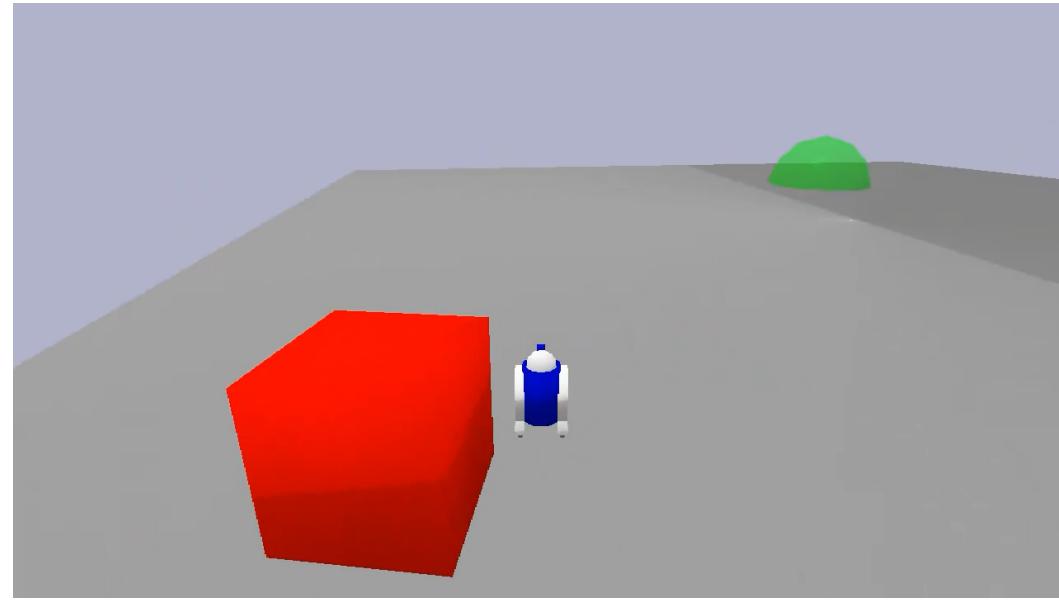


Figure 26: Task 18 of the OMNI-EPIC run presented in Section 5.

```

4340
4341
4342
4343
4344 Success:
4345 The task is successfully completed when the cube is entirely within
4346 the target zone.
4347
4348 Rewards:
4349 - The robot receives a survival reward for each time step it remains
4350   on the platform, encouraging steady progress.
4351 - The robot is rewarded based on how much it reduces the distance to
4352   the cube, incentivizing interaction with the cube.
4353 - The robot is rewarded based on how much it reduces the distance
4354   between the cube and the target zone, incentivizing pushing the
4355   cube towards the goal.
4356 - A large reward is given when the cube reaches the target zone.
4357
4358 Termination:
4359 The episode terminates if the robot falls off the platform or if the
4360   cube reaches the target zone.
4361 """
4362
4363 def __init__(self):
4364     super().__init__()
4365     self.platform_size = [50.0, 50.0, 0.1]
4366     self.platform_position = [0.0, 0.0, 0.0]
4367     self.platform_id = self.create_box(mass=0.0, half_extents=[self.
4368         platform_size[0] / 2, self.platform_size[1] / 2, self.
4369         platform_size[2] / 2], position=self.platform_position, color
4370         =[0.8, 0.8, 0.8, 1.0])
4371     self._p.changeDynamics(bodyUniqueId=self.platform_id, linkIndex
4372         =-1, lateralFriction=0.8, restitution=0.5)
4373     self.cube_size = [2.0, 2.0, 2.0]
4374     self.cube_mass = 5.0
4375     self.cube_id = self.create_box(mass=self.cube_mass, half_extents
4376         =[self.cube_size[0] / 2, self.cube_size[1] / 2, self.
4377         cube_size[2] / 2], position=[0.0, 0.0, 0.0], color=[1.0, 0.0,
4378             0.0, 1.0])
4379     self.target_zone_radius = 3.0

```

```

4374     self.target_zone_id = self.create_sphere(mass=0.0, radius=self.
4375         target_zone_radius, collision=False, position=[0.0, 0.0,
4376             0.0], color=[0.0, 1.0, 0.0, 0.5])
4377     self.robot_position_init = [0.0, 0.0, self.platform_position[2] +
4378         self.platform_size[2] / 2 + self.robot.links['base'].
4379         position_init[2]]
4380
4381     def create_box(self, mass, half_extents, position, color):
4382         collision_shape_id = self._p.createCollisionShape(shapeType=self._p.GEOM_BOX, halfExtents=half_extents)
4383         visual_shape_id = self._p.createVisualShape(shapeType=self._p.GEOM_BOX, halfExtents=half_extents, rgbaColor=color)
4384         return self._p.createMultiBody(baseMass=mass,
4385             baseCollisionShapeIndex=collision_shape_id,
4386             baseVisualShapeIndex=visual_shape_id, basePosition=position)
4387
4388     def create_sphere(self, mass, radius, collision, position, color):
4389         if collision:
4390             collision_shape_id = self._p.createCollisionShape(shapeType=self._p.GEOM_SPHERE, radius=radius)
4391             visual_shape_id = self._p.createVisualShape(shapeType=self._p.GEOM_SPHERE, radius=radius, rgbaColor=color)
4392             return self._p.createMultiBody(baseMass=mass,
4393                 baseCollisionShapeIndex=collision_shape_id,
4394                 baseVisualShapeIndex=visual_shape_id, basePosition=
4395                 position)
4396         else:
4397             visual_shape_id = self._p.createVisualShape(shapeType=self._p.GEOM_SPHERE, radius=radius, rgbaColor=color)
4398             return self._p.createMultiBody(baseMass=mass,
4399                 baseVisualShapeIndex=visual_shape_id, basePosition=
4400                 position)
4401
4402     def get_object_position(self, object_id):
4403         return np.asarray(self._p.getBasePositionAndOrientation(object_id)[0])
4404
4405     def get_distance_between_objects(self, object1_id, object2_id):
4406         object1_position = self.get_object_position(object1_id)
4407         object2_position = self.get_object_position(object2_id)
4408         return np.linalg.norm(object1_position[:2] - object2_position[:2])
4409
4410     def reset(self):
4411         observation = super().reset()
4412         self._p.resetBasePositionAndOrientation(self.robot.robot_id, self.
4413             .robot_position_init, self.robot.links['base'].
4414             orientation_init)
4415         cube_x = np.random.uniform(self.platform_position[0] - self.
4416             platform_size[0] / 2 + self.cube_size[0] / 2, self.
4417             platform_position[0] + self.platform_size[0] / 2 - self.
4418             cube_size[0] / 2)
4419         cube_y = np.random.uniform(self.platform_position[1] - self.
4420             platform_size[1] / 2 + self.cube_size[1] / 2, self.
4421             platform_position[1] + self.platform_size[1] / 2 - self.
4422             cube_size[1] / 2)
4423         self._p.resetBasePositionAndOrientation(self.cube_id, [cube_x,
4424             cube_y, self.platform_position[2] + self.platform_size[2] / 2
4425             + self.cube_size[2] / 2], [0.0, 0.0, 0.0, 1.0])
4426         target_zone_x = np.random.uniform(self.platform_position[0] -
4427             self.platform_size[0] / 2 + self.target_zone_radius, self.
4428             platform_position[0] + self.platform_size[0] / 2 - self.
4429             target_zone_radius)
4430         target_zone_y = np.random.uniform(self.platform_position[1] -
4431             self.platform_size[1] / 2 + self.target_zone_radius, self.

```

```

4428     platform_position[1] + self.platform_size[1] / 2 - self.
4429     target_zone_radius)
4430     self._p.resetBasePositionAndOrientation(self.target_zone_id, [
4431         target_zone_x, target_zone_y, self.platform_position[2] +
4432         self.platform_size[2] / 2], [0.0, 0.0, 0.0, 1.0])
4433     return observation
4434
4435     def step(self, action):
4436         self.distance_robot_to_cube = self.get_distance_between_objects(
4437             self.robot.robot_id, self(cube_id))
4438         self.distance_cube_to_target_zone = self.
4439             get_distance_between_objects(self(cube_id, self.
4440                 target_zone_id)
4441         observation, reward, terminated, truncated, info = super().step(
4442             action)
4443         return (observation, reward, terminated, truncated, info)
4444
4445     def get_task_rewards(self, action):
4446         new_distance_robot_to_cube = self.get_distance_between_objects(
4447             self.robot.robot_id, self(cube_id))
4448         new_distance_cube_to_target_zone = self.
4449             get_distance_between_objects(self(cube_id, self.
4450                 target_zone_id)
4451         survival = 1.0
4452         reach_cube = (self.distance_robot_to_cube -
4453             new_distance_robot_to_cube) / self.dt
4454         push_cube_to_target_zone = (self.distance_cube_to_target_zone -
4455             new_distance_cube_to_target_zone) / self.dt
4456         cube_in_target_zone = 10.0 if new_distance_cube_to_target_zone <
4457             self.target_zone_radius else 0.0
4458         return {'survival': survival, 'reach_cube': reach_cube, '
4459             push_cube_to_target_zone': push_cube_to_target_zone, '
4460             cube_in_target_zone': cube_in_target_zone}
4461
4462     def get_terminated(self, action):
4463         is_off_platform = self.robot.links['base'].position[2] < self.
4464             platform_position[2]
4465         is_cube_in_target_zone = self.get_distance_between_objects(self.
4466             cube_id, self.target_zone_id) < self.target_zone_radius
4467         return is_off_platform or is_cube_in_target_zone
4468
4469     def get_success(self):
4470         is_cube_in_target_zone = self.get_distance_between_objects(self.
4471             cube_id, self.target_zone_id) < self.target_zone_radius
4472         return is_cube_in_target_zone

```

### Task 19

```

4468 import numpy as np
4469 from oped.envs.r2d2.base import R2D2Env
4470
4471 class Env(R2D2Env):
4472     """
4473         Push a cube into a target zone in a large open arena.
4474
4475     Description:
4476     - The environment consists of a large flat arena measuring 50 x 50
4477         meters.
4478     - A cube with dimensions of 2 meters in size and a mass of 5 kg is
4479         placed at a random location within the arena.
4480     - A target zone with a radius of 3 meters is also randomly placed
4481         within the arena. The target zone has no collision, allowing the
4482         cube to be pushed into it without obstruction.
4483     - The robot is initialized at a random position within the arena,
4484         facing the positive x-axis.

```

```

4482
4483
4484
4485
4486
4487
4488
4489
4490
4491
4492
4493
4494
4495
4496
4497
4498
4499
4500
4501
4502
4503
4504
4505
4506
4507
4508
4509
4510
4511
4512
4513
4514
4515
4516
4517
4518
4519
4520
4521
4522
4523
4524
4525
4526
4527
4528
4529
4530
4531
4532
4533
4534
4535

```

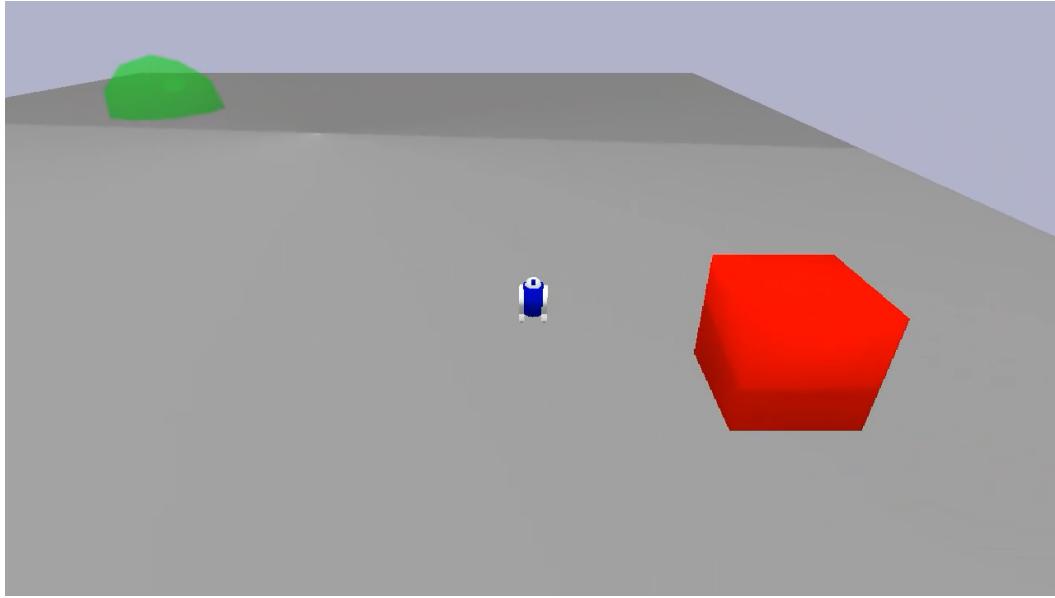


Figure 27: Task 19 of the OMNI-EPIC run presented in Section 5.

```

Success:
The task is successfully completed when the robot pushes the cube
into the target zone.

Rewards:
– The robot receives a survival reward at each time step to encourage
steady progress.
– The robot is rewarded for decreasing its distance to the cube,
encouraging it to approach and interact with the cube.
– The robot is rewarded for pushing the cube towards the target zone,
with additional rewards for getting the cube into the target
zone.
– The robot is rewarded for remaining within the arena, avoiding
falling off the platform.

Termination:
The episode terminates if the robot falls off the platform. The
episode does not terminate if the cube is in the target zone.
"""

def __init__(self):
    super().__init__()
    self.arena_size = [50.0, 50.0, 0.1]
    self.arena_position = [0.0, 0.0, 0.0]
    self.arena_id = self.create_box(mass=0.0, half_extents=[self.
        arena_size[0] / 2, self.arena_size[1] / 2, self.arena_size[2]
        / 2], position=self.arena_position, color=[0.5, 0.5, 0.5,
        1.0])
    self._p.changeDynamics(bodyUniqueId=self.arena_id, linkIndex=-1,
        lateralFriction=0.8, restitution=0.5)
    self.cube_size = [2.0, 2.0, 2.0]
    self.cube_mass = 5.0
    self.cube_id = self.create_box(mass=self.cube_mass, half_extents
        =[self.cube_size[0] / 2, self.cube_size[1] / 2, self.
        cube_size[2] / 2], position=[0.0, 0.0, 0.0], color=[1.0, 0.0,
        0.0, 1.0])
    self.target_zone_radius = 3.0

```

```

4536     self.target_zone_id = self.create_sphere(mass=0.0, radius=self.
4537         target_zone_radius, collision=False, position=[0.0, 0.0,
4538             0.0], color=[0.0, 1.0, 0.0, 0.5])
4539
4540     def create_box(self, mass, half_extents, position, color):
4541         collision_shape_id = self._p.createCollisionShape(shapeType=self.
4542             _p.GEOM_BOX, halfExtents=half_extents)
4543         visual_shape_id = self._p.createVisualShape(shapeType=self._p.
4544             GEOM_BOX, halfExtents=half_extents, rgbaColor=color)
4545         return self._p.createMultiBody(baseMass=mass,
4546             baseCollisionShapeIndex=collision_shape_id,
4547             baseVisualShapeIndex=visual_shape_id, basePosition=position)
4548
4549     def create_sphere(self, mass, radius, collision, position, color):
4550         if collision:
4551             collision_shape_id = self._p.createCollisionShape(shapeType=
4552                 self._p.GEOM_SPHERE, radius=radius)
4553             visual_shape_id = self._p.createVisualShape(shapeType=self._p.
4554                 .GEOM_SPHERE, radius=radius, rgbaColor=color)
4555             return self._p.createMultiBody(baseMass=mass,
4556                 baseCollisionShapeIndex=collision_shape_id,
4557                 baseVisualShapeIndex=visual_shape_id, basePosition=
4558                     position)
4559         else:
4560             visual_shape_id = self._p.createVisualShape(shapeType=self._p.
4561                 .GEOM_SPHERE, radius=radius, rgbaColor=color)
4562             return self._p.createMultiBody(baseMass=mass,
4563                 baseVisualShapeIndex=visual_shape_id, basePosition=
4564                     position)
4565
4566     def get_object_position(self, object_id):
4567         return np.asarray(self._p.getBasePositionAndOrientation(object_id
4568             )[0])
4569
4570     def get_distance_to_object(self, object_id):
4571         object_position = self.get_object_position(object_id)
4572         robot_position = self.robot.links['base'].position
4573         return np.linalg.norm(object_position[:2] - robot_position[:2])
4574
4575     def reset(self):
4576         observation = super().reset()
4577         cube_x_init = np.random.uniform(low=-self.arena_size[0] / 2 +
4578             self.cube_size[0] / 2, high=self.arena_size[0] / 2 - self.
4579             cube_size[0] / 2)
4580         cube_y_init = np.random.uniform(low=-self.arena_size[1] / 2 +
4581             self.cube_size[1] / 2, high=self.arena_size[1] / 2 - self.
4582             cube_size[1] / 2)
4583         self._p.resetBasePositionAndOrientation(self.cube_id, [
4584             cube_x_init, cube_y_init, self.arena_position[2] + self.
4585             arena_size[2] / 2 + self.cube_size[2] / 2], [0.0, 0.0, 0.0,
4586             1.0])
4587         target_zone_x = np.random.uniform(low=-self.arena_size[0] / 2 +
4588             self.target_zone_radius, high=self.arena_size[0] / 2 - self.
4589             target_zone_radius)
4590         target_zone_y = np.random.uniform(low=-self.arena_size[1] / 2 +
4591             self.target_zone_radius, high=self.arena_size[1] / 2 - self.
4592             target_zone_radius)
4593         self.target_zone_position = [target_zone_x, target_zone_y, self.
4594             arena_position[2] + self.arena_size[2] / 2]
4595         self._p.resetBasePositionAndOrientation(self.target_zone_id, self
4596             .target_zone_position, [0.0, 0.0, 0.0, 1.0])
4597         robot_x_init = np.random.uniform(low=-self.arena_size[0] / 2 +
4598             self.robot.links['base'].position_init[0], high=self.
4599             arena_size[0] / 2 - self.robot.links['base'].position_init
4600             [0])

```

```

4590     robot_y_init = np.random.uniform(low=-self.arena_size[1] / 2 +
4591                                         self.robot.links['base'].position_init[1], high=self.
4592                                         arena_size[1] / 2 - self.robot.links['base'].position_init
4593                                         [1])
4594     self._p.resetBasePositionAndOrientation(self.robot.robot_id, [
4595         robot_x_init, robot_y_init, self.arena_position[2] + self.
4596         arena_size[2] / 2 + self.robot.links['base'].position_init
4597         [2]], self.robot.links['base'].orientation_init)
4598     return observation
4599
4600     def step(self, action):
4601         self.distance_to_cube = self.get_distance_to_object(self.cube_id)
4602         self.distance_cube_to_target_zone = self.get_distance_to_object(
4603             self.target_zone_id)
4604         self.cube_position = self.get_object_position(self.cube_id)
4605         observation, reward, terminated, truncated, info = super().step(
4606             action)
4607         return (observation, reward, terminated, truncated, info)
4608
4609     def get_task_rewards(self, action):
4610         new_distance_to_cube = self.get_distance_to_object(self.cube_id)
4611         new_distance_cube_to_target_zone = self.get_distance_to_object(
4612             self.target_zone_id)
4613         new_cube_position = self.get_object_position(self.cube_id)
4614         survival = 1.0
4615         reach_cube = (self.distance_to_cube - new_distance_to_cube) /
4616                     self.dt
4617         push_cube = (self.distance_cube_to_target_zone -
4618                     new_distance_cube_to_target_zone) / self.dt
4619         if new_distance_cube_to_target_zone < self.target_zone_radius:
4620             push_cube += 5.0
4621         in_arena = 1.0 if abs(self.robot.links['base'].position[0]) <
4622                     self.arena_size[0] / 2 and abs(self.robot.links['base'].
4623                     position[1]) < self.arena_size[1] / 2 else -1.0
4624         return {'survival': survival, 'reach_cube': reach_cube, '
4625             push_cube': push_cube, 'in_arena': in_arena}
4626
4627     def get_terminated(self, action):
4628         return abs(self.robot.links['base'].position[0]) > self.
4629             arena_size[0] / 2 or abs(self.robot.links['base'].position
4630             [1]) > self.arena_size[1] / 2
4631
4632     def get_success(self):
4633         cube_distance_to_target_zone = self.get_distance_to_object(self.
4634             target_zone_id)
4635         return cube_distance_to_target_zone < self.target_zone_radius

```

## Task 21

```

4632     import numpy as np
4633     from oped.envs.r2d2.base import R2D2Env
4634
4635     class Env(R2D2Env):
4636         """
4637             Push a domino to start a chain reaction.
4638
4639             Description:
4640             - The environment consists of a large platform measuring 1000 x 10 x
4641               0.1 meters.
4642             - The robot is initialized at a fixed position on the platform.
4643             - A domino with dimensions 0.5 x 2 x 4 meters and a mass of 5 kg is
4644               positioned on the platform, 5 meters away from the robot.
4645             - The dominos are spaced by 3 meters, and positioned to create a
4646               chain reaction.

```

```

4644
4645
4646
4647
4648
4649
4650
4651
4652
4653
4654
4655
4656
4657
4658
4659
4660
4661
4662
4663
4664
4665
4666
4667
4668
4669
4670
4671
4672
4673
4674
4675
4676
4677
4678
4679
4680
4681
4682
4683
4684
4685
4686
4687
4688
4689
4690
4691
4692
4693
4694
4695
4696
4697

```

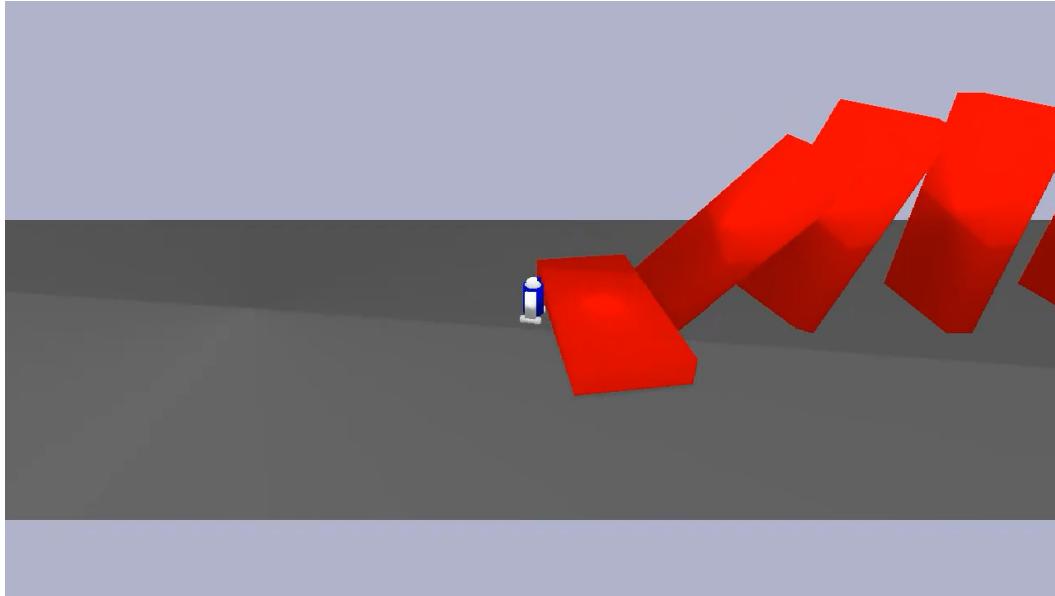


Figure 28: Task 21 of the OMNI-EPIC run presented in Section 5.

```

-- The task of the robot is to push the domino to start a chain
reaction.

Success:
The task is successfully completed when the robot pushes the domino
and starts the chain reaction.

Rewards:
-- The robot is rewarded for forward progress towards the domino,
encouraging it to approach and push the domino.
-- The robot is rewarded for the number of dominos that fall,
incentivizing it to initiate the chain reaction.
-- The robot is penalized for falling off the platform, ensuring it
stays on the platform.

Termination:
The task terminates immediately if the robot falls off the platform.
"""

def __init__(self):
    super().__init__()
    self.platform_size = [1000.0, 10.0, 0.1]
    self.platform_position = [0.0, 0.0, 0.0]
    self.platform_id = self.create_box(mass=0.0, half_extents=[self.
        platform_size[0] / 2, self.platform_size[1] / 2, self.
        platform_size[2] / 2], position=self.platform_position, color
        =[0.5, 0.5, 0.5, 1.0])
    self._p.changeDynamics(bodyUniqueId=self.platform_id, linkIndex
        =-1, lateralFriction=0.8, restitution=0.5)
    self.domino_size = [0.5, 2.0, 4.0]
    self.domino_mass = 5.0
    self.domino_spacing = 3.0
    self.num_dominos = 10
    self.domino_ids = []
    for i in range(self.num_dominos):
        domino_position = [self.platform_position[0] + 5.0 + i * self.
            domino_spacing, self.platform_position[1], self.
            platform_size[2] / 2]
        self.domino_ids.append(self.create_box(mass=self.domino_mass, half_extents=[self.
            domino_size[0] / 2, self.domino_size[1] / 2, self.
            domino_size[2] / 2], position=domino_position, color
            =[0.5, 0.5, 0.5, 1.0], dynamic=False))

```

```

4698     platform_position[2] + self.platform_size[2] / 2 + self.
4699     domino_size[2] / 2]
4700     domino_id = self.create_box(mass=self.domino_mass,
4701         half_extents=[self.domino_size[0] / 2, self.domino_size
4702             [1] / 2, self.domino_size[2] / 2], position=
4703             domino_position, color=[1.0, 0.0, 0.0, 1.0])
4704     self.domino_ids.append(domino_id)
4705     self.robot_position_init = [self.platform_position[0], self.
4706         platform_position[1], self.platform_position[2] + self.
4707             platform_size[2] / 2 + self.robot.links['base'].position_init
4708             [2]]
4709
4710     def create_box(self, mass, half_extents, position, color):
4711         collision_shape_id = self._p.createCollisionShape(shapeType=self.
4712             _p.GEOM_BOX, halfExtents=half_extents)
4713         visual_shape_id = self._p.createVisualShape(shapeType=self._p.
4714             GEOM_BOX, halfExtents=half_extents, rgbaColor=color)
4715         return self._p.createMultiBody(baseMass=mass,
4716             baseCollisionShapeIndex=collision_shape_id,
4717             baseVisualShapeIndex=visual_shape_id, basePosition=position)
4718
4719     def get_object_position(self, object_id):
4720         return np.asarray(self._p.getBasePositionAndOrientation(object_id
4721             )[0])
4722
4723     def get_distance_to_object(self, object_id):
4724         object_position = self.get_object_position(object_id)
4725         robot_position = self.robot.links['base'].position
4726         return np.linalg.norm(object_position[:2] - robot_position[:2])
4727
4728     def reset(self):
4729         observation = super().reset()
4730         self._p.resetBasePositionAndOrientation(self.robot.robot_id, self
4731             .robot_position_init, self.robot.links['base'].
4732             orientation_init)
4733         for i, domino_id in enumerate(self.domino_ids):
4734             domino_position = [self.platform_position[0] + 5.0 + i * self
4735                 .domino_spacing, self.platform_position[1], self.
4736                 platform_position[2] + self.platform_size[2] / 2 + self.
4737                     domino_size[2] / 2]
4738             self._p.resetBasePositionAndOrientation(domino_id,
4739                 domino_position, [0.0, 0.0, 0.0, 1.0])
4740
4741         return observation
4742
4743     def step(self, action):
4744         self.distance_to_first_domino = self.get_distance_to_object(self.
4745             domino_ids[0])
4746         self.num_fallen_dominos = sum([self._p.getBaseVelocity(domino_id)
4747             [0][2] < -0.1 for domino_id in self.domino_ids])
4748         observation, reward, terminated, truncated, info = super().step(
4749             action)
4750
4751         return (observation, reward, terminated, truncated, info)
4752
4753     def get_task_rewards(self, action):
4754         new_distance_to_first_domino = self.get_distance_to_object(self.
4755             domino_ids[0])
4756         new_num_fallen_dominos = sum([self._p.getBaseVelocity(domino_id)
4757             [0][2] < -0.1 for domino_id in self.domino_ids])
4758         forward_progress = (self.distance_to_first_domino -
4759             new_distance_to_first_domino) / self.dt
4760         domino_fall_reward = (new_num_fallen_dominos - self.
4761             num_fallen_dominos) * 10.0
4762
4763         return {'forward_progress': forward_progress, 'domino_fall_reward'
4764             : domino_fall_reward}

```

```

4752     def get_terminated(self, action):
4753         return self.robot.links['base'].position[2] < self.
4754             platform_position[2]
4755
4756     def get_success(self):
4757         return any([self._p.getBaseVelocity(domino_id)[0][2] < -0.1 for
4758             domino_id in self.domino_ids])

```

### Task 23

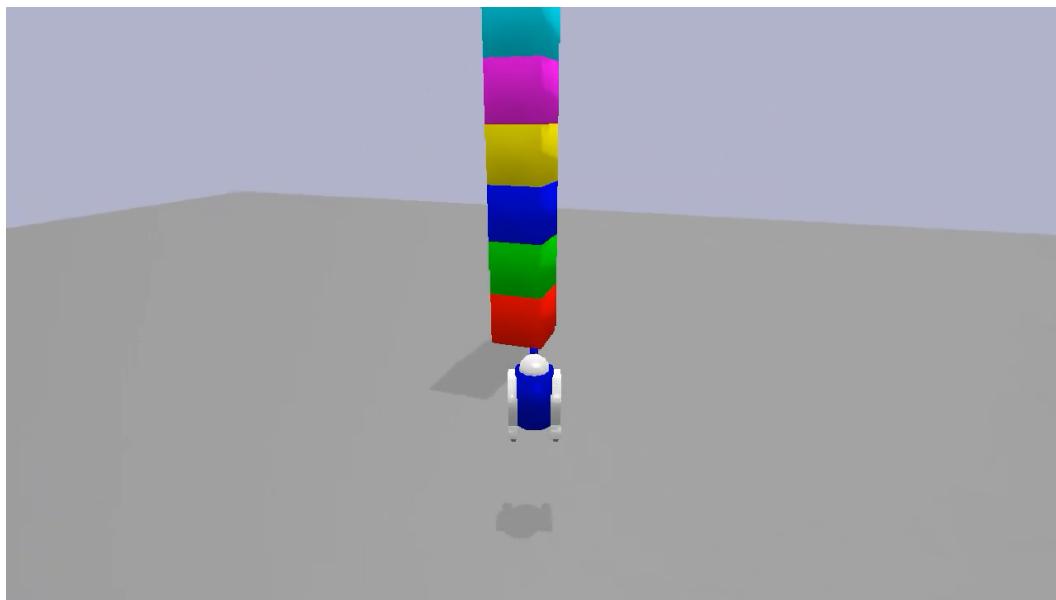


Figure 29: Task 23 of the OMNI-EPIC run presented in Section 5.

```

4781
4782
4783
4784 import numpy as np
4785 from oped.envs.r2d2.base import R2D2Env
4786
4787 class Env(R2D2Env):
4788     """
4789     Task: Destroy a tower of blocks
4790
4791     Description:
4792     - The environment consists of a large flat platform measuring 50
4793         meters by 50 meters.
4794     - A tower of 10 blocks is placed at a random location on the platform
4795         . Each block is a cube measuring 1 meter by 1 meter by 1 meter
4796         and weighs 5 kg. The blocks are of different colors to make the
4797         environment visually rich.
4798     - The robot is initialized at a fixed position on the platform,
4799         facing the tower.
4800
4801     The task of the robot is to destroy the tower by pushing the blocks
4802         until all blocks are knocked over and no longer stacked.
4803
4804     Success:
4805     The task is successfully completed when all blocks are knocked over
4806         and none are stacked on top of each other.
4807
4808     Rewards:
4809     - The robot receives a survival reward for each time step it remains
4810         on the platform, encouraging steady progress.

```

```

4806     - The robot is rewarded for reducing the height of the tower,
4807       incentivizing the robot to knock over the blocks.
4808     - A large reward is given when all blocks are knocked over and none
4809       are stacked.
4810
4811 Termination:
4812 The episode terminates if the robot falls off the platform or if all
4813   blocks are knocked over and none are stacked.
4814 """
4815
4816 def __init__(self):
4817     super().__init__()
4818     self.platform_size = [50.0, 50.0, 0.1]
4819     self.platform_position = [0.0, 0.0, 0.0]
4820     self.platform_id = self.create_box(mass=0.0, half_extents=[self.
4821         platform_size[0] / 2, self.platform_size[1] / 2, self.
4822         platform_size[2] / 2], position=self.platform_position, color
4823         =[0.8, 0.8, 0.8, 1.0])
4824     self._p.changeDynamics(bodyUniqueId=self.platform_id, linkIndex
4825         =-1, lateralFriction=0.8, restitution=0.5)
4826     self.block_size = [1.0, 1.0, 1.0]
4827     self.block_mass = 5.0
4828     self.num_blocks = 10
4829     self.block_ids = []
4830     self.block_colors = [[1.0, 0.0, 0.0, 1.0], [0.0, 1.0, 0.0, 1.0],
4831         [0.0, 0.0, 1.0, 1.0], [1.0, 1.0, 0.0, 1.0], [1.0, 0.0, 1.0,
4832             1.0], [0.0, 1.0, 1.0, 1.0], [0.5, 0.5, 0.5, 1.0], [1.0, 0.5,
4833             0.0, 1.0], [0.5, 0.0, 0.5, 1.0], [0.0, 0.5, 0.5, 1.0]]
4834     for i in range(self.num_blocks):
4835         block_id = self.create_box(mass=self.block_mass, half_extents
4836             =[self.block_size[0] / 2, self.block_size[1] / 2, self.
4837                 block_size[2] / 2], position=[0.0, 0.0, 0.0], color=self.
4838                 block_colors[i])
4839         self.block_ids.append(block_id)
4840     self.robot_position_init = [0.0, 0.0, self.platform_position[2] +
4841         self.platform_size[2] / 2 + self.robot.links['base'].
4842             position_init[2]]
4843
4844 def create_box(self, mass, half_extents, position, color):
4845     collision_shape_id = self._p.createCollisionShape(shapeType=self.
4846         _p.GEOM_BOX, halfExtents=half_extents)
4847     visual_shape_id = self._p.createVisualShape(shapeType=self._p.
4848         GEOM_BOX, halfExtents=half_extents, rgbaColor=color)
4849     return self._p.createMultiBody(baseMass=mass,
4850         baseCollisionShapeIndex=collision_shape_id,
4851         baseVisualShapeIndex=visual_shape_id, basePosition=position)
4852
4853 def get_object_position(self, object_id):
4854     return np.asarray(self._p.getBasePositionAndOrientation(object_id
4855         )[0])
4856
4857 def get_distance_between_objects(self, object1_id, object2_id):
4858     object1_position = self.get_object_position(object1_id)
4859     object2_position = self.get_object_position(object2_id)
4860     return np.linalg.norm(object1_position[:2] - object2_position
4861         [:2])
4862
4863 def reset(self):
4864     observation = super().reset()
4865     self._p.resetBasePositionAndOrientation(self.robot.robot_id, self
4866         .robot_position_init, self.robot.links['base'].
4867         orientation_init)
4868     tower_x = np.random.uniform(self.platform_position[0] - self.
4869         platform_size[0] / 2 + self.block_size[0] / 2, self.

```

```

4860     platform_position[0] + self.platform_size[0] / 2 - self.
4861     block_size[0] / 2)
4862     tower_y = np.random.uniform(self.platform_position[1] - self.
4863       platform_size[1] / 2 + self.block_size[1] / 2, self.
4864       platform_position[1] + self.platform_size[1] / 2 - self.
4865       block_size[1] / 2)
4866     for i, block_id in enumerate(self.block_ids):
4867       self._p.resetBasePositionAndOrientation(block_id, [tower_x,
4868         tower_y, self.platform_position[2] + self.platform_size
4869         [2] / 2 + (i + 0.5) * self.block_size[2]], [0.0, 0.0,
4870         0.0, 1.0])
4871   return observation
4872
4873 def step(self, action):
4874   self.tower_height = self.calculate_tower_height()
4875   observation, reward, terminated, truncated, info = super().step(
4876     action)
4877   return (observation, reward, terminated, truncated, info)
4878
4879 def calculate_tower_height(self):
4880   heights = [self.get_object_position(block_id)[2] for block_id in
4881     self.block_ids]
4882   return max(heights) - min(heights)
4883
4884 def get_task_rewards(self, action):
4885   new_tower_height = self.calculate_tower_height()
4886   survival = 1.0
4887   reduce_tower_height = (self.tower_height - new_tower_height) /
4888     self.dt
4889   all_blocks_knocked_over = 10.0 if new_tower_height <= self.
4890     block_size[2] else 0.0
4891   return {'survival': survival, 'reduce_tower_height':
4892     reduce_tower_height, 'all_blocks_knocked_over':
4893     all_blocks_knocked_over}
4894
4895 def get_terminated(self, action):
4896   is_off_platform = self.robot.links['base'].position[2] < self.
4897     platform_position[2]
4898   all_blocks_knocked_over = self.calculate_tower_height() <= self.
4899     block_size[2]
4900   return is_off_platform or all_blocks_knocked_over
4901
4902 def get_success(self):
4903   all_blocks_knocked_over = self.calculate_tower_height() <= self.
4904     block_size[2]
4905   return all_blocks_knocked_over

```

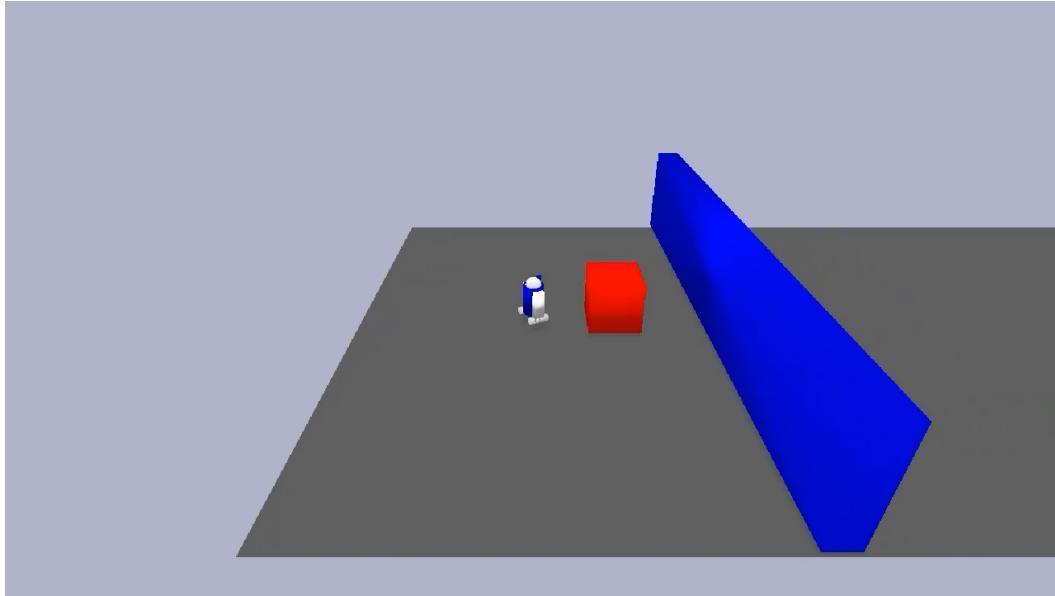
## Task 25

```

4902 import numpy as np
4903 from oped.envs.r2d2.base import R2D2Env
4904
4905 class Env(R2D2Env):
4906   """
4907     Push a box next to a wall to climb on top of it.
4908
4909     Description:
4910     - The environment consists of a large platform measuring 1000 x 10 x
4911       0.1 meters.
4912     - The robot is initialized at a fixed position on the platform.
4913     - A wall with dimensions 0.5 x 10 x 2 meters is positioned on the
4914       platform, 5 meters away from the robot.
4915     - A box with dimensions 1 x 1 x 1 meters (5 kg in mass) is positioned
4916       on the platform, 3 meters away from the robot.

```

4914  
4915  
4916  
4917  
4918  
4919  
4920  
4921  
4922  
4923  
4924  
4925  
4926  
4927  
4928  
4929  
4930  
4931  
4932  
4933



4934 Figure 30: Task 25 of the OMNI-EPIC run presented in Section 5.  
4935  
4936  
4937

- The task of the robot is first to push the box against the wall.  
Then, the robot must climb on top of the box and jump on top of the wall.

Success:

The task is successfully completed if the robot climbs on top of the box and then jumps on top of the wall.

Rewards:

- The robot receives a reward for each time step it remains on the platform.
- The robot is rewarded for decreasing its distance to the box, encouraging it to approach and interact with the box.
- The robot is rewarded for pushing the box towards the wall.
- The robot is rewarded for climbing on top of the box and an additional reward for jumping on top of the wall.

Termination:

The episode terminates if the robot falls off the platform. The episode does not terminate if the robot successfully jumps on top of the wall.

\*\*\*

```
4956
4957     def __init__(self):
4958         super().__init__()
4959         self.robot_position_init = [0.0, 0.0, 0.0]
4960         self.platform_size = [1000.0, 10.0, 0.1]
4961         self.platform_position = [self.robot_position_init[0] + self.
4962             platform_size[0] / 2 - 2.0, self.robot_position_init[1], self.
4963             .robot_position_init[2] - self.platform_size[2] / 2]
4964         self.platform_id = self.create_box(mass=0.0, half_extents=[self.
4965             platform_size[0] / 2, self.platform_size[1] / 2, self.
4966             platform_size[2] / 2], position=self.platform_position, color
4967             =[0.5, 0.5, 0.5, 1.0])
4968         self._p.changeDynamics(bodyUniqueId=self.platform_id, linkIndex
4969             =-1, lateralFriction=0.8, restitution=0.5)
4970         self.wall_size = [0.5, 10.0, 2.0]
```

```

4968     self.wall_position = [self.robot_position_init[0] + 5.0, self.
4969         platform_position[1], self.platform_position[2] + self.
4970             platform_size[2] / 2 + self.wall_size[2] / 2]
4971     self.wall_id = self.create_box(mass=0.0, half_extents=[self.
4972         wall_size[0] / 2, self.wall_size[1] / 2, self.wall_size[2] /
4973             2], position=self.wall_position, color=[0.0, 0.0, 1.0, 1.0])
4974     self.box_size = [1.0, 1.0, 1.0]
4975     self.box_mass = 5.0
4976     self.box_position_init = [self.robot_position_init[0] + 3.0, self.
4977         .platform_position[1], self.platform_position[2] + self.
4978             platform_size[2] / 2 + self.box_size[2] / 2]
4979     self.box_id = self.create_box(mass=self.box_mass, half_extents=[
4980         self.box_size[0] / 2, self.box_size[1] / 2, self.box_size[2] /
4981             2], position=self.box_position_init, color=[1.0, 0.0, 0.0,
4982                 1.0])
4983
4984     def create_box(self, mass, half_extents, position, color):
4985         collision_shape_id = self._p.createCollisionShape(shapeType=self.
4986             _p.GEOM_BOX, halfExtents=half_extents)
4987         visual_shape_id = self._p.createVisualShape(shapeType=self._p.
4988             GEOM_BOX, halfExtents=half_extents, rgbaColor=color)
4989         return self._p.createMultiBody(baseMass=mass,
4990             baseCollisionShapeIndex=collision_shape_id,
4991             baseVisualShapeIndex=visual_shape_id, basePosition=position)
4992
4993     def get_object_position(self, object_id):
4994         return np.asarray(self._p.getBasePositionAndOrientation(object_id
4995             )[0])
4996
4997     def get_distance_to_object(self, object_id):
4998         object_position = self.get_object_position(object_id)
4999         robot_position = self.robot.links['base'].position
5000         return np.linalg.norm(object_position[:2] - robot_position[:2])
5001
5002     def reset(self):
5003         observation = super().reset()
5004         self._p.resetBasePositionAndOrientation(self.box_id, self.
5005             box_position_init, [0.0, 0.0, 0.0, 1.0])
5006         self._p.resetBasePositionAndOrientation(self.robot.robot_id, [
5007             self.robot_position_init[0], self.robot_position_init[1],
5008                 self.robot_position_init[2] + self.robot.links['base'].
5009                     position_init[2]], self.robot.links['base'].orientation_init)
5010         return observation
5011
5012     def step(self, action):
5013         self.distance_to_box = self.get_distance_to_object(self.box_id)
5014         self.box_position = self.get_object_position(self.box_id)
5015         self.robot_on_box = len(self._p.getContactPoints(bodyA=self.robot.
5016             .robot_id, bodyB=self.box_id)) > 0
5017         self.robot_on_wall = len(self._p.getContactPoints(bodyA=self.
5018             robot.robot_id, bodyB=self.wall_id)) > 0
5019         observation, reward, terminated, truncated, info = super().step(
5020             action)
5021         return (observation, reward, terminated, truncated, info)
5022
5023     def get_task_rewards(self, action):
5024         new_distance_to_box = self.get_distance_to_object(self.box_id)
5025         new_box_position = self.get_object_position(self.box_id)
5026         new_robot_on_box = len(self._p.getContactPoints(bodyA=self.robot.
5027             .robot_id, bodyB=self.box_id)) > 0
5028         new_robot_on_wall = len(self._p.getContactPoints(bodyA=self.robot.
5029             .robot_id, bodyB=self.wall_id)) > 0
5030         survival = 1.0
5031         reach_box = (self.distance_to_box - new_distance_to_box) / self.
5032             dt

```

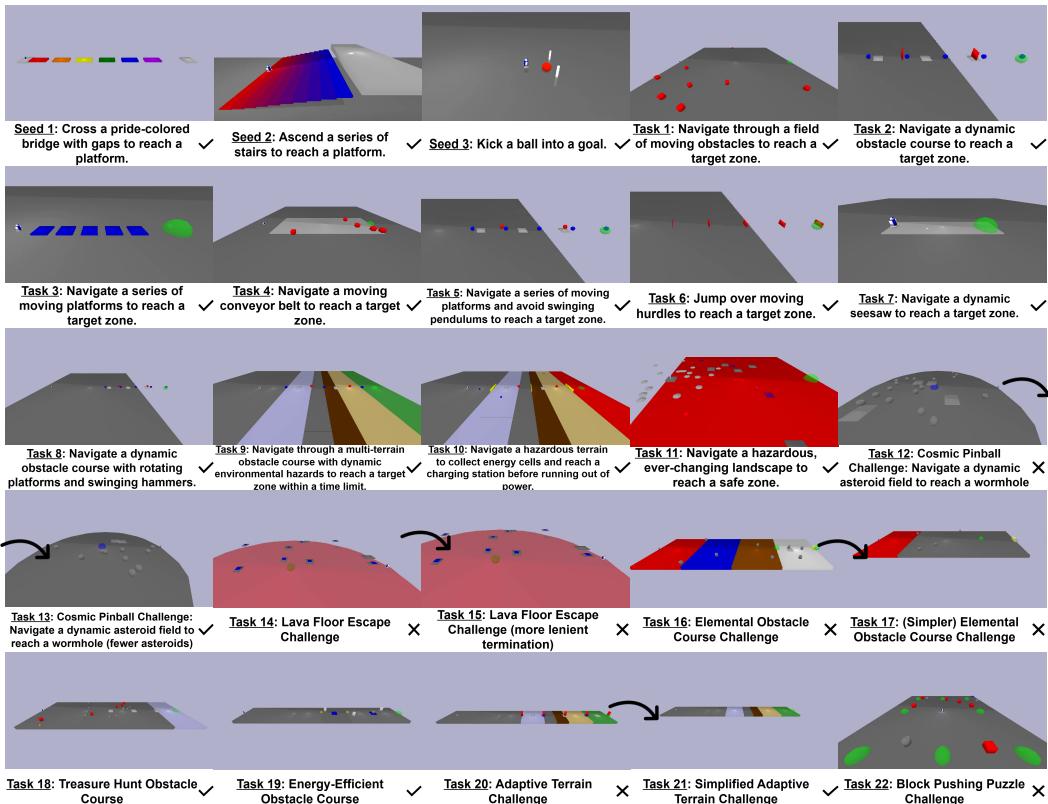
```

5022     push_box = (new_box_position[0] - self.box_position[0]) / self.dt
5023     climb_on_box = 5.0 if new_robot_on_box and (not self.robot_on_box
5024         ) else 0.0
5025     jump_on_wall = 10.0 if new_robot_on_wall and (not self.
5026         robot_on_wall) else 0.0
5027     return {'survival': survival, 'reach_box': reach_box, 'push_box':
5028         push_box, 'climb_on_box': climb_on_box, 'jump_on_wall':
5029         jump_on_wall}
5030
5031     def get_terminated(self, action):
5032         return self.robot.links['base'].position[2] < self.
5033             platform_position[2]
5034
5035     def get_success(self):
5036         return len(self._p.getContactPoints(bodyA=self.robot.robot_id,
5037             bodyB=self.wall_id)) > 0
5038
5039
5040
5041
5042
5043
5044
5045
5046
5047
5048
5049
5050
5051
5052
5053
5054
5055
5056
5057
5058
5059
5060
5061
5062
5063
5064
5065
5066
5067
5068
5069
5070
5071
5072
5073
5074
5075

```

## 5076 F ADDITIONAL SHORT RUNS WITH LEARNING

5078 Although the initial seed tasks in the task archive remain the same across repeated runs, we observe  
 5079 significant differences in the generated environments. This suggests that OMNI-EPIC’s ability to  
 5080 generate diverse learnable tasks is not heavily influenced by the initial set of environments.



5108 **Figure 31: Additional Short Run with Learning (Run 2).** This figure shows another instance of  
 5109 the short run experiment under the same settings as in Figure 3. It demonstrates the progression of  
 5110 tasks generated by OMNI-EPIC and the learning outcomes of the RL agent. Checkmarks indicate  
 5111 successful learning, crosses indicate failures, and arrows show iterations on failed tasks.

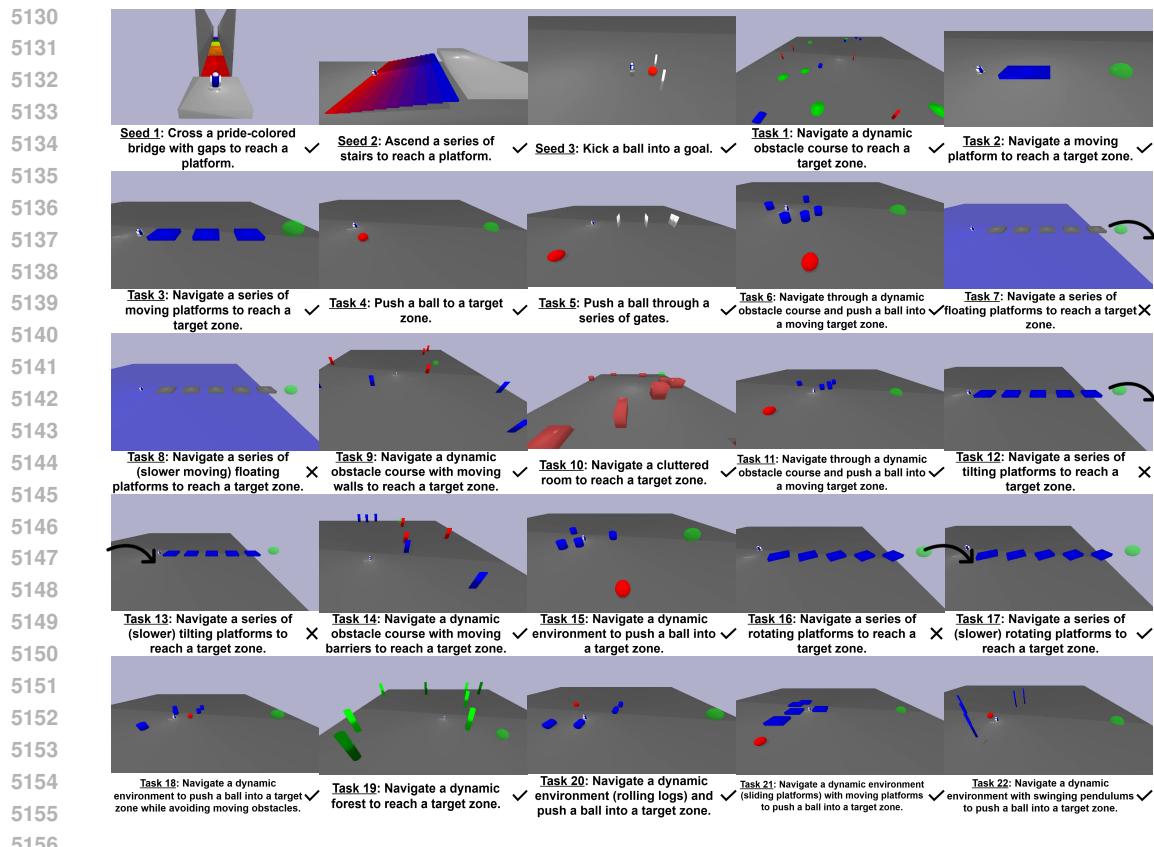


Figure 32: **Additional Short Run with Learning (Run 3).** This figure shows another instance of the short run experiment under the same settings as in Figure 3. It demonstrates the progression of tasks generated by OMNI-EPIC and the learning outcomes of the RL agent. Checkmarks indicate successful learning, crosses indicate failures, and arrows show iterations on failed tasks.

5161  
5162  
5163  
5164  
5165  
5166  
5167  
5168  
5169  
5170  
5171  
5172  
5173  
5174  
5175  
5176  
5177  
5178  
5179  
5180  
5181  
5182  
5183

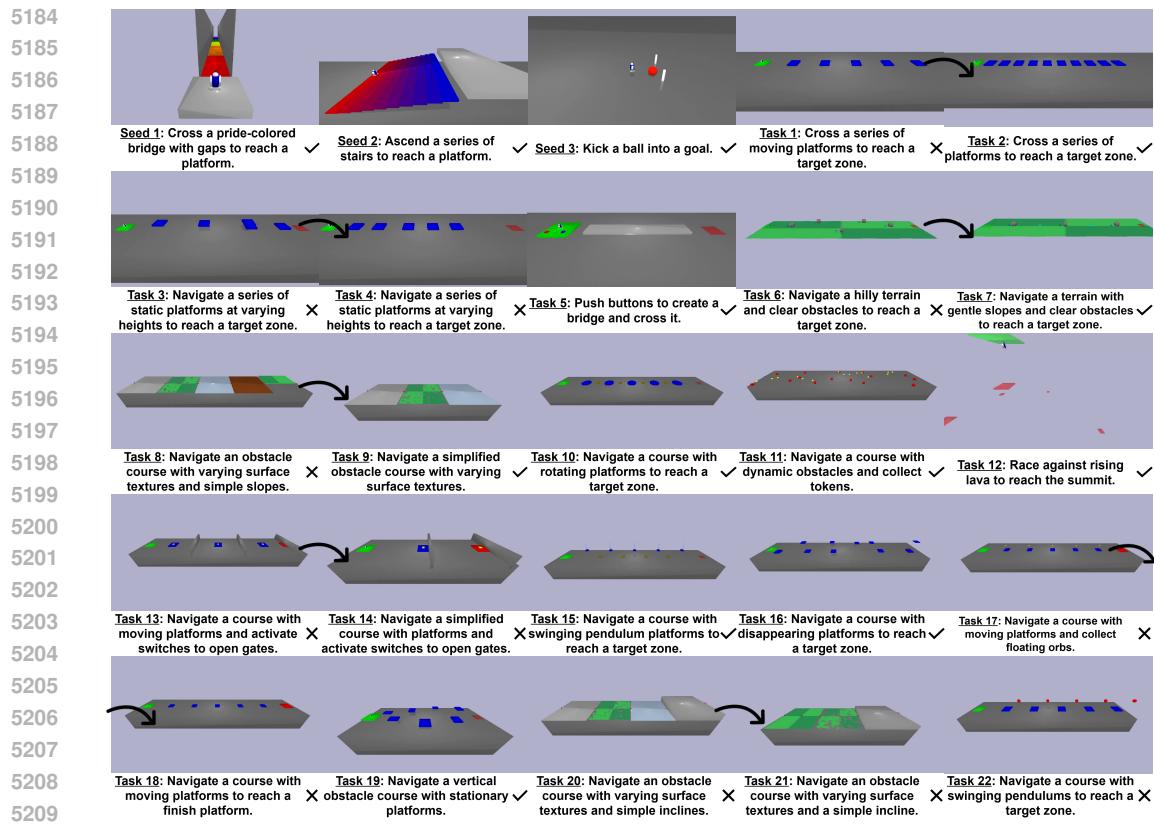


Figure 33: **Additional Short Run with Learning (Run 4).** This figure shows another instance of the short run experiment under the same settings as in Figure 3. It demonstrates the progression of tasks generated by OMNI-EPIC and the learning outcomes of the RL agent. Checkmarks indicate successful learning, crosses indicate failures, and arrows show iterations on failed tasks.

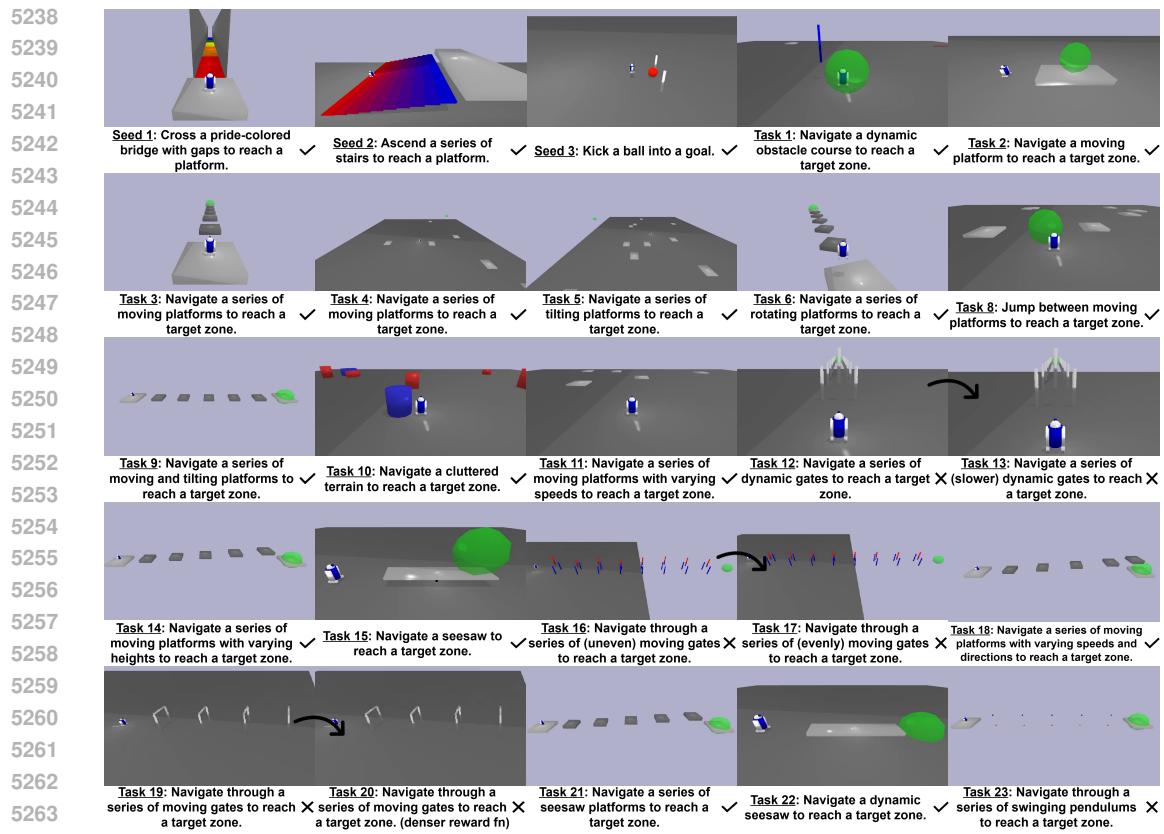


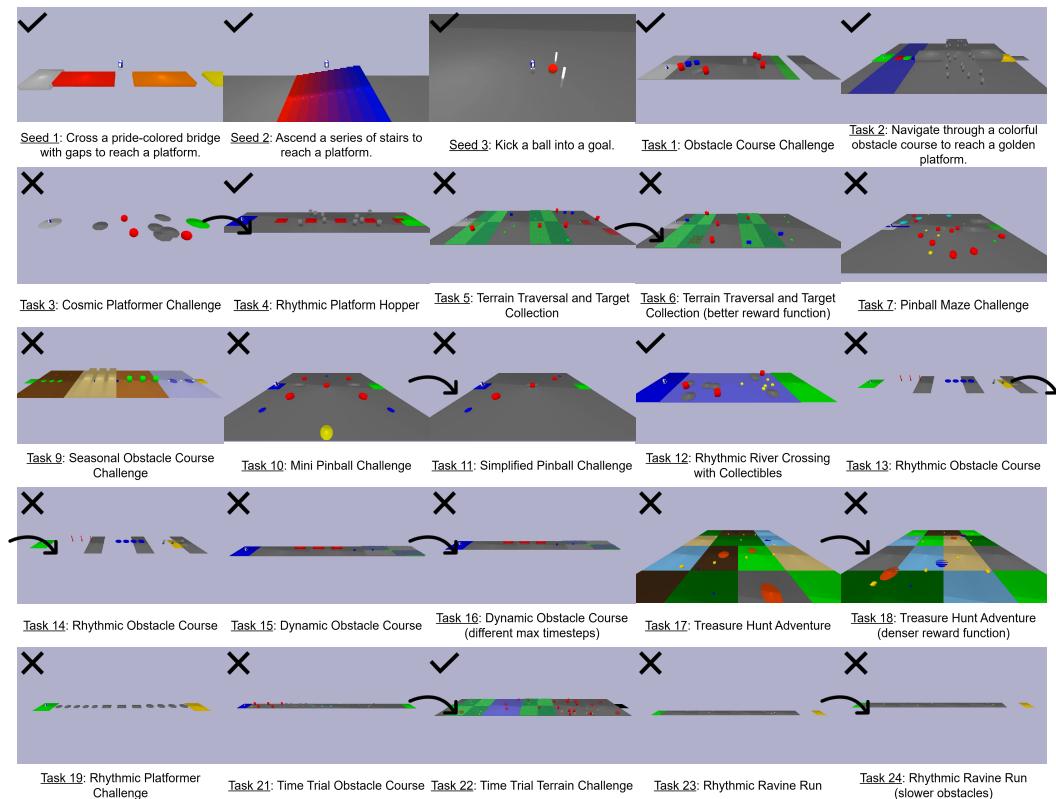
Figure 34: **Additional Short Run with Learning (Run 5).** This figure shows another instance of the short run experiment under the same settings as in Figure 3. It demonstrates the progression of tasks generated by OMNI-EPIC and the learning outcomes of the RL agent. Checkmarks indicate successful learning, crosses indicate failures, and arrows show iterations on failed tasks.

## 5292 G ABLATIONS ON SHORT RUNS WITH LEARNING

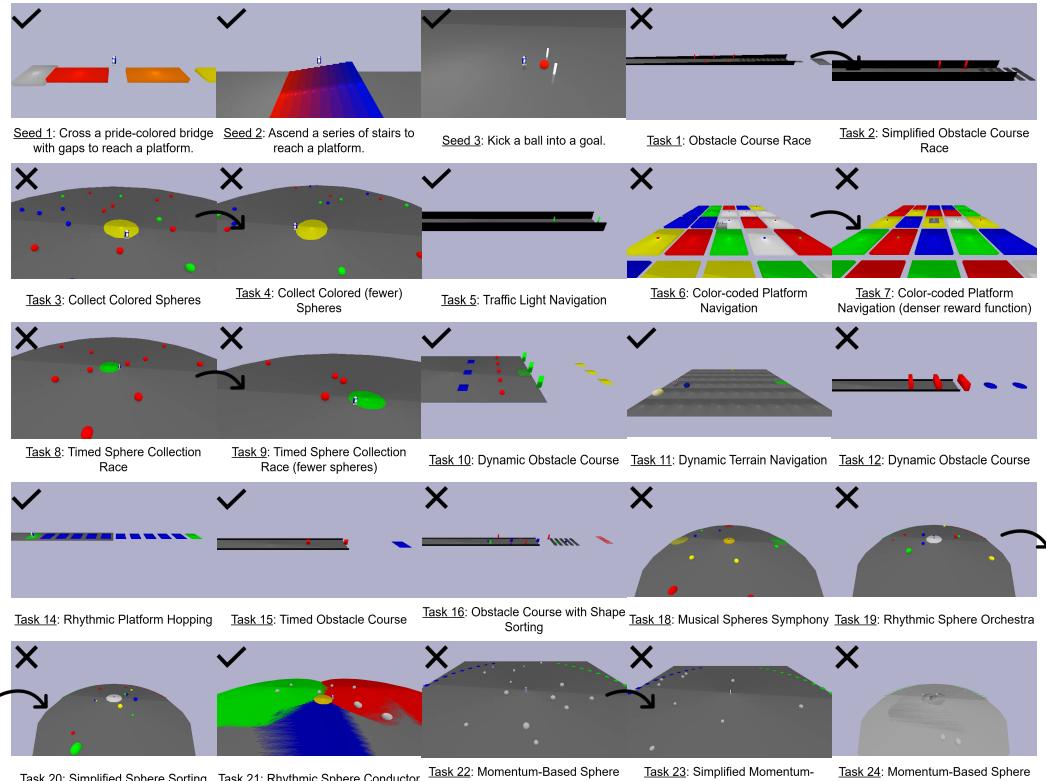
### 5294 G.1 WITHOUT TRANSFER LEARNING

5296 We conducted an experiment in which we ran OMNI-EPIC (i.e., with transfer learning between tasks)  
 5297 on 25 tasks with 5 replications and compared it to training policies from scratch (i.e., without transfer  
 5298 learning between tasks). We allocated the same total number of training steps (2 million) to the RL  
 5299 agent for each task. OMNI-EPIC achieves a significantly higher success rate, with a median of 70.6%  
 5300 (CI: 61.4 - 74.1) than training from scratch, with a median of 57.9% (CI: 44.6 - 65.5) ( $p < 0.05$ ,  
 5301 Mann-Whitney U test). This demonstrates that the OMNI-EPIC agents build upon previously learned  
 5302 skills, creating a curriculum of increasing difficulty.

### 5303 G.2 UNIFORM SAMPLING EXAMPLES FROM THE ARCHIVE

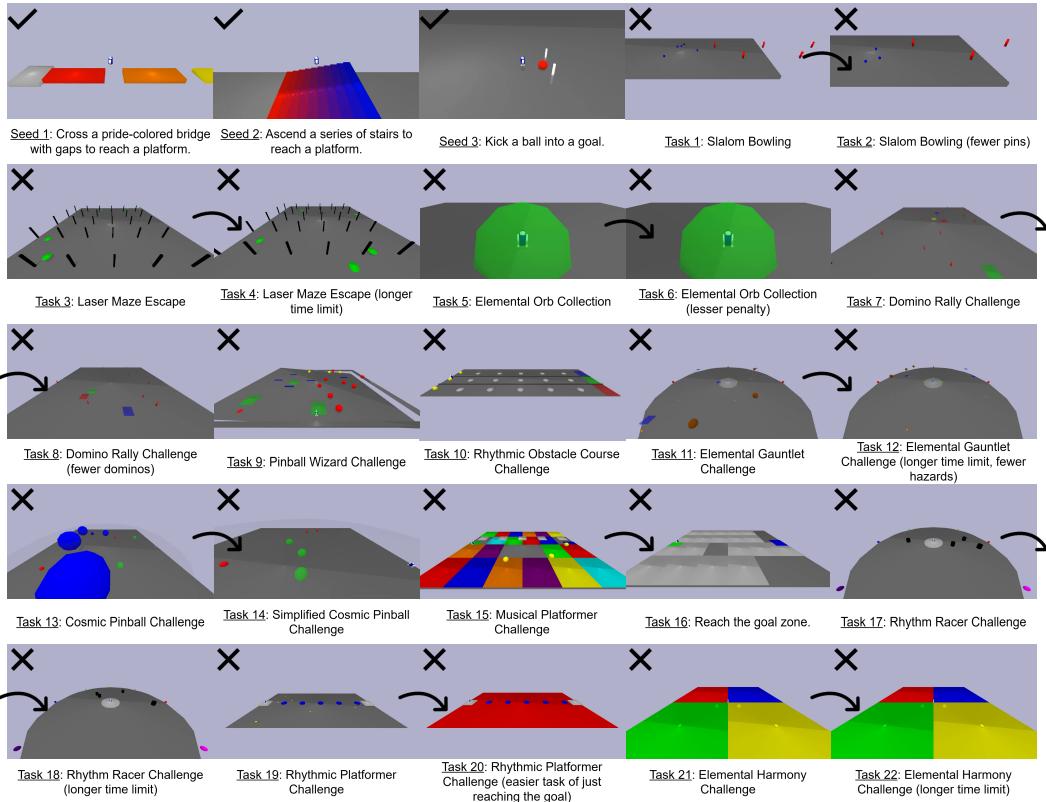


5329 **Figure 35: Ablation of short run with learning, by sampling in-context examples uniformly.** This  
 5330 figure shows an ablation of the short run with learning experiment. The same settings as Figure 3 are  
 5331 used, except that, instead of using the most similar tasks as in-context examples for the task generator,  
 5332 we uniformly sample from the archive. Checkmarks indicate successful learning, crosses indicate  
 5333 failures, and arrows show iterations on failed tasks. Using uniform sampling for in-context examples  
 5334 results in learning only 5 of the generated tasks, much fewer than in Figure 3, where the most similar  
 5335 tasks are used as in-context examples.

5346 G.3 WITHOUT FAILED EXAMPLES  
5347

5374 **Figure 36: Ablation of short run with learning, without failed examples input to the task**  
 5375 **generator.** This figure shows an ablation of the short run with learning experiment. The same  
 5376 settings as Figure 3 are used, except that no failed examples (only successful ones) are given to  
 5377 the task generator. Checkmarks indicate successful learning, crosses indicate failures, and arrows  
 5378 show iterations on failed tasks. We see that similar tasks are regenerated, even though the RL  
 5379 agent previously failed to learn them due to unsuitable reward function or incorrect environment  
 5380 configuration. Not giving the task generator examples of tasks that were attempted but failed results  
 5381 in learning only 7 of the generated tasks, much fewer than in Figure 3, where both successful and  
 5382 failed examples are used as input.

5383  
5384  
5385  
5386  
5387  
5388  
5389  
5390  
5391  
5392  
5393  
5394  
5395  
5396  
5397  
5398  
5399

5400 G.4 WITHOUT LEARNING PROGRESS  
5401

5429 **Figure 37: Ablation of short run with learning, without learning progress notions in the task  
5430 generator's prompt.** This figure shows an ablation of the short run with learning experiment. The  
5431 same settings as Figure 3 are used, except that the task generator's prompt does not include any  
5432 notion of learning progress. Checkmarks indicate successful learning, crosses indicate failures, and  
5433 arrows show iterations on failed tasks. Without incorporating a notion of learning progress in the task  
5434 generator's prompt, no tasks were successfully learned, which is much fewer than in Figure 3, where  
5435 the task generator's prompt includes both notions of interestingness and learning progress.

5436 **System Prompt:**  
5437

5438 You are an expert in reinforcement learning. Your goal is to help a robot  
5439 master a diverse set of interesting tasks in simulation using  
5440 PyBullet. You will be provided with the list of tasks that the robot  
5441 has successfully learned, along with their corresponding environment  
5442 code, and the list of tasks that the robot has attempted but failed  
5443 to learn, along with their corresponding environment code. Your  
5444 objective is to decide the next task for the robot, selecting one  
5445 that is interesting and novel.

5446 **Instructions:**

- 5447 – The next task should be interesting:
  - 5448 – Novel and creative compared to the tasks the robot has already tried.
  - 5449 – Useful according to humans.
  - 5450 – Design rich environments with a large number of diverse objects and terrains, and with a clear task for the robot to execute.
  - 5451 – The task should be fun or engaging to watch. You can draw inspiration from real-world tasks or video games. Be creative!
- 5452 – Be specific in the task description:
  - 5453 – State clearly what the task of the robot is.

- 5454           – Define clearly what the success condition is.  
 5455           – Define clearly what are the different reward and penalty components  
 5456           ·  
 5457           – Define clearly what the termination conditions are. If the reward  
 5458            components include a survival reward, ensure the episode only  
 5459            terminates when the agent fails the task.  
 5460           – The task should not take too long to complete.  
 5461           – The robot can push objects around but lacks the ability to grab, pick  
 5462            up, carry, or stack objects. Don't suggest tasks that involve these  
 5463            skills.  
 5464           – Don't suggest tasks that require the robot to navigate through a maze.  
 5465           – If the task involves navigating a terrain with obstacles, make sure  
 5466            that the robot can not go around the obstacles.  
 5467           – If the task involves a target zone, make sure that the collision of the  
 5468            target zone is set to False.  
 5469           – Return only the task description, not the environment code.  
 5470           – Ensure that the designs pose no harm to humans and align with human  
 5471            values and ethics.

5470       Robot description:  
 5471       {ROBOT\_DESC}

5472       Desired format:

5473       Reasoning for what the next task should be:  
 5474       <reasoning>

5475       Next task description:

5476       \"\\\""  
 5477       <task description>  
 5478       \"\\\""

5479  
 5480  
 5481  
 5482  
 5483  
 5484  
 5485  
 5486  
 5487  
 5488  
 5489  
 5490  
 5491  
 5492  
 5493  
 5494  
 5495  
 5496  
 5497  
 5498  
 5499  
 5500  
 5501  
 5502  
 5503  
 5504  
 5505  
 5506  
 5507

## 5508 H HUMAN EVALUATION SETUP

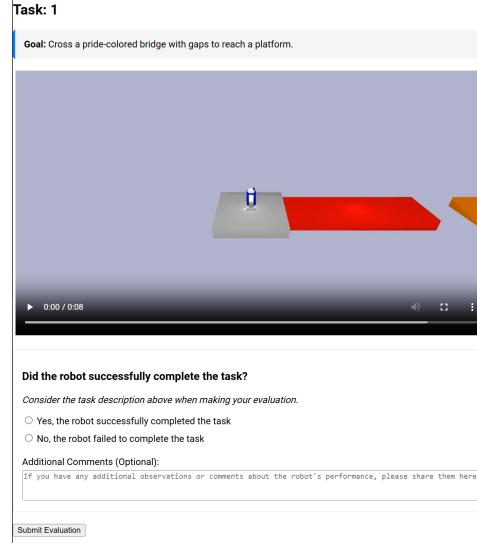
5509  
 5510 To evaluate the alignment between human judgment and the automated success detector, we conducted  
 5511 a study with 50 participants. The goal was to assess whether human evaluators agreed with the  
 5512 success detector's assessments of task completion by the robot. Each participant reviewed videos of a  
 5513 robot attempting various tasks, alongside the corresponding task descriptions. Below are the detailed  
 5514 instructions and setup used for the evaluation.

5515  
 5516 View videos of a robot attempting specific tasks, each accompanied by a task  
 5517 description.  
 5518

- 5519  
 5520
  - 5521 1. **Read the Task Description.**
  - 5522 2. **Watch the Video.**
  - 5523 3. **Decide:** Did the robot successfully complete the task as described?
  - 5524 4. **Respond:** Choose "**Successfully Completed**" or "**Did Not Complete**".

5525  
 5526 **Notes:**  
 5527 Base your judgment solely on the task description and the video.  
 Be objective and honest.

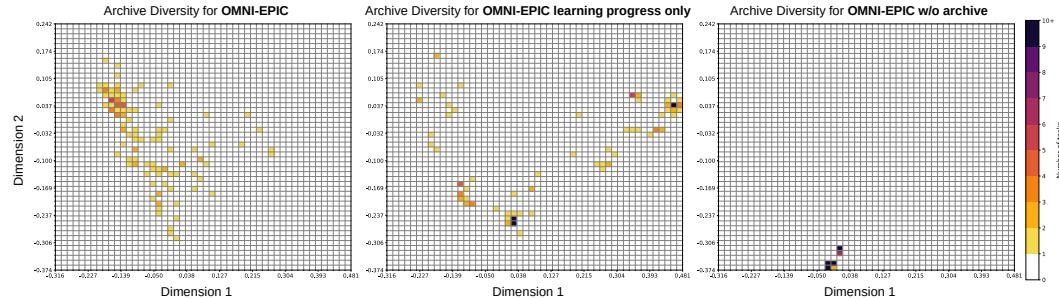
5528 **Example:**  
 5529 Task: "Cross a bridge with moving segments to reach a platform."  
 If the robot reaches the platform without falling, answer: "Successfully  
 5530 Completed".  
 If the robot falls off the bridge before reaching the platform, answer: "Did Not  
 5531 Complete".  
 5532



5533 Figure 38: **Human evaluation setup.** (Left) Instructions given to participants. (Right) Annotation  
 5534 interface used by participants to annotate the data.

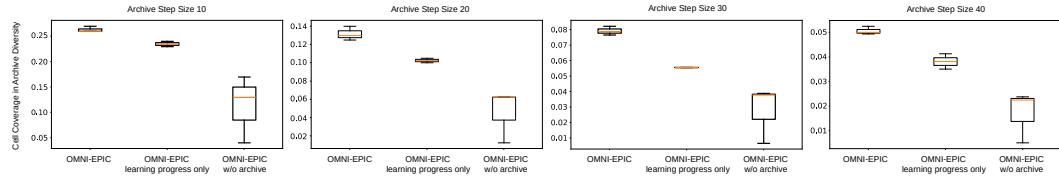
5535 The human evaluation study was conducted using the `cloudresearch.com` platform. Participants  
 5536 were compensated \$0.25 for each response (estimated hourly rate of \$15.00/hour, as each response  
 5537 took less than 1 minute), ensuring fair and ethical payment for their time and effort. The study  
 5538 aimed to collect accurate and unbiased assessments to compare with the success detector's automated  
 5539 evaluations.

## 5562 I SUPPLEMENTARY MATERIALS FOR QUANTITATIVE RESULTS

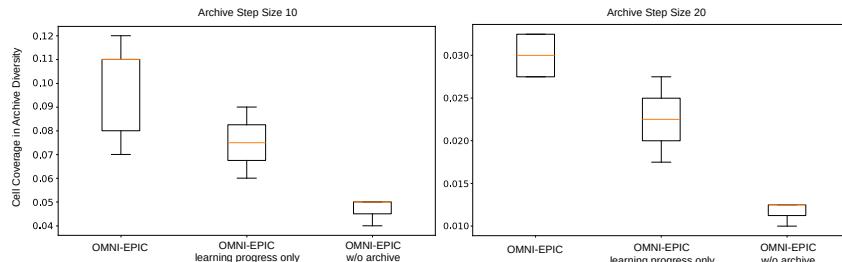


5565  
5566  
5567  
5568  
5569  
5570  
5571  
5572  
5573  
5574  
5575  
5576  
5577  
5578  
5579  
5580  
5581  
5582  
5583  
5584  
5585  
5586  
5587  
5588  
5589  
5590  
5591  
5592  
5593  
5594  
5595  
5596  
5597  
5598  
5599  
5600  
5601  
5602  
5603  
5604  
5605  
5606  
5607  
5608  
5609  
5610  
5611  
5612  
5613  
5614  
5615

**Figure 39: Archive diversity results.** Archive diversity plots of long runs with simulated learning of OMNI-EPIC and the controls. Fewer tasks fall into the same discretized cells for OMNI-EPIC than OMNI-EPIC Learning Progress only or OMNI-EPIC w/o archive. The substantial difference between the left and center plots is more easily observed in Figure 4.



**Figure 40: Cell coverage of archive diversity plots with different discretization levels for long runs with simulated learning.** This figure is similar to Figure 4, which uses an archive discretization level of 50, but here we present cell coverage results for archive diversity plots with discretization levels of [10, 20, 30, 40] across methods on long runs with simulated learning. OMNI-EPIC consistently achieves significantly higher cell coverage compared to the controls, even across different archive discretization levels ( $p < 0.05$ , Mann-Whitney U test).



**Figure 41: Cell coverage of archive diversity plots with different discretization levels for short runs with learning.** This figure is similar to Figure 4, which uses an archive discretization level of 50 for long run with simulated learning, but here we present cell coverage results for archive diversity plots with discretization levels of [10, 20] across methods on short runs with learning. While we see similar trends as Figure 4 and Figure 40, the differences between methods are not always statistically significant (not all  $p < 0.05$ , Mann-Whitney U test). This is due to the shorter training runs, as the effects of OMNI-EPIC become more pronounced over longer runs.

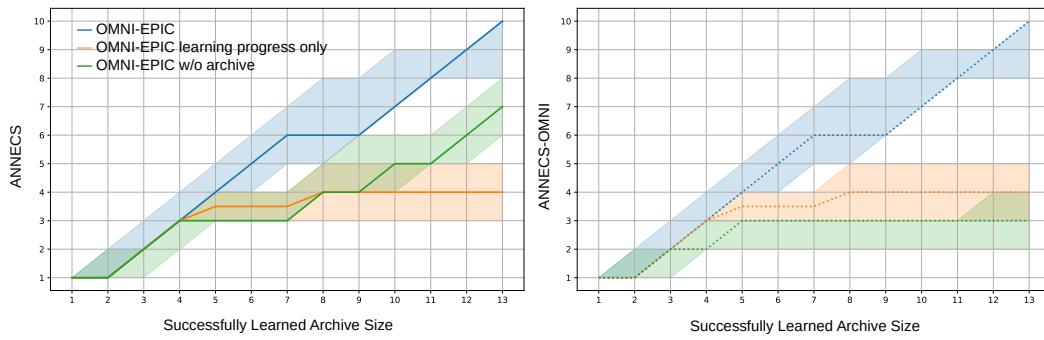


Figure 42: **(Left) ANNECS and (Right) ANNECS-OMNI results.** Short runs with RL training by OMNI-EPIC and the controls are repeated five times. Darker lines are median values, shaded regions are 95% confidence intervals. OMNI-EPIC significantly outperforms the controls on both metrics. There is no difference between ANNECS and ANNECS-OMNI for OMNI-EPIC, indicating that all tasks learned by OMNI-EPIC are considered interesting.

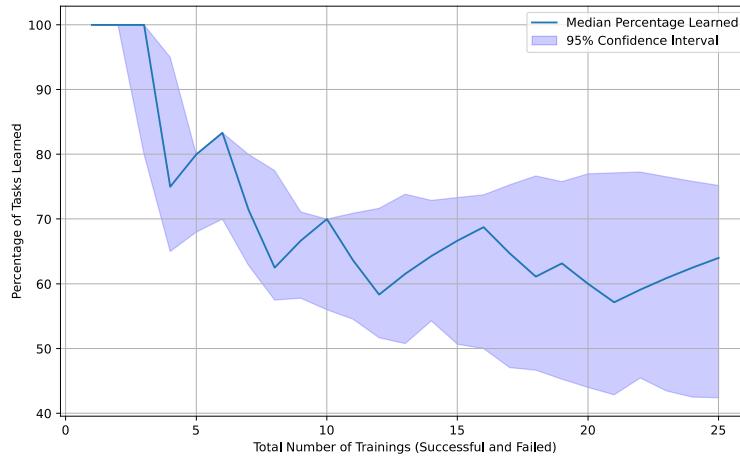


Figure 43: **Percentage of tasks learned over all attempted tasks in short runs with learning experiments.** Attempted tasks include both all tasks that were attempted with RL training. Short runs with RL training by OMNI-EPIC are repeated five times. Darker lines are median values, shaded regions are 95% confidence intervals. The decreasing percentage of learned tasks over time indicates that OMNI-EPIC generates tasks of increasing difficulty over time.

---

**J SELECTING THE MOST SIMILAR TASKS**

To generate new tasks that are both relevant and challenging, it is essential to select tasks from the archive as a reference. This selection process ensures that the generated tasks build upon the agent’s prior knowledge and skills. OMNI-EPIC’s process of selecting the most similar tasks, given a query task, takes inspiration from Lewis et al. (2020). The first step is to embed all tasks (both their natural language descriptions and environment code) within the archive into a high-dimensional vector space using a pretrained language encoder. This embedding allows us to represent each task as a dense vector that captures its semantic meaning and characteristics. Once all tasks are embedded, we can efficiently compare the similarity between tasks using cosine similarity. For a given query task, we retrieve a predefined number of tasks from the archive that exhibit the closest cosine similarity to the query’s embedding, ensuring that the selected tasks are the most relevant and similar to the current learning context.

A potential limitation of our approach is the possibility of cyclic behavior where the task generator alternates between generating task types that are consistently rejected due to their similarity to existing tasks in the archive. While this scenario is unlikely in practice due to the stochastic nature of task generation and the diversity of the environment distribution, it cannot be completely ruled out. Furthermore, as context lengths increase with advancements in FM capabilities, the likelihood of such cyclic behavior would diminish even more.

5670  
5671  
5672  
5673  
5674  
5675  
5676  
5677  
5678  
5679  
5680  
5681  
5682  
5683  
5684  
5685  
5686  
5687  
5688  
5689  
5690  
5691  
5692  
5693  
5694  
5695  
5696  
5697  
5698  
5699  
5700  
5701  
5702  
5703  
5704  
5705  
5706  
5707  
5708  
5709  
5710  
5711  
5712  
5713  
5714  
5715  
5716  
5717  
5718  
5719  
5720  
5721  
5722  
5723

5724 **K HYPERPARAMETERS**

5725

5726

5727

Table 1: OMNI-EPIC hyperparameters

Component	Parameter	Value
General	Max number of iterations	5
	Embedding method	OpenAI (text-embedding-3-small)
Task Generator	Number of learned examples	5
	Number of failed examples	5
	Client	Anthropic
	Model	Claude 3 Opus (claude-3-opus-20240229)
Environment Generator	Temperature	0
	Number of few-shot examples	5
	Client	Anthropic
	Model	Claude 3 Opus (claude-3-opus-20240229)
Post-Generation MoI	Temperature	0
	Number of similar tasks	10
	Client	OpenAI
	Model	GPT-4o (gpt-4o-2024-05-13)
Success Detector	Temperature	0
	Client	OpenAI
	Model	GPT-4o (gpt-4o-2024-05-13)
Task Reflection	Temperature	0
	Max number of iterations	1
	Number of few-shot examples	5
	Client	OpenAI
Model	Model	GPT-4o (gpt-4o-2024-05-13)
	Temperature	0

Table 2: DreamerV3 hyperparameters

Parameter	Value
Total time steps	$2 \times 10^6$
Replay buffer size	$10^6$
Batch size	16
Batch length	64
Discount factor	0.997
Learning rate	$3 \times 10^{-4}$

5770 **Compute Resources**

5771 For each task, the R2D2 agent is trained for approximately 1 hour using 2 NVIDIA RTX 6000 Ada  
5772 Generation GPUs and 32 CPU cores.

5773

5774  
5775  
5776  
5777

5778 L PROMPTS  
 5779

5780 L.1 TASK GENERATOR  
 5781

5782 System Prompt:  
 5783

You are an expert in curriculum learning and reinforcement learning. Your goal is to help a robot master a diverse set of interesting tasks in simulation using PyBullet. You will be provided with the list of tasks that the robot has successfully learned, along with their corresponding environment code, and the list of tasks that the robot has attempted but failed to learn, along with their corresponding environment code. Your objective is to decide the next task for the robot, selecting one that will maximize learning effectiveness based on its past successes and failures.

5792 Instructions:  
 5793

- The next task should be learnable:
  - Not too difficult for the robot to learn given its current skill set.
  - Realistic for the robot based on its description.
  - Possible to complete in simulation in PyBullet.
- The next task should be interesting, i.e., either:
  - Novel compared to the tasks the robot has already learned. You can either add complexity gradually on an existing task or design a radically novel task from scratch.
  - Useful according to humans, making it worth learning.
  - Creative or surprising.
  - Optionally, the task can be fun and engaging to watch.
- Be specific in the task description:
  - State clearly what the task of the robot is.
  - If the task involves objects, be specific about their positions and orientations relative to the robot. Be careful to avoid collisions between objects or with the robot when you decide on the initial positions.
  - If the task involves dynamically moving objects, be specific about their movement.
- You can draw inspiration from real-world tasks or video games. Be creative!
- The task should not take too long to complete.
- The robot can push objects around but lacks the ability to grab, pick up, carry, or stack objects. Don't suggest tasks that involve these skills.
- Don't suggest tasks that require the robot to navigate through a maze.
- Return only the task description, not the environment code.
- Ensure that the designs pose no harm to humans and align with human values and ethics.

5818 Robot description:  
 5819 {ROBOT\_DESC}  
 5820

5821 Desired format:

5822 Reasoning for what the next task should be:

5823 <reasoning>

5824 Next task description:

5825 """

5826 <task description>

5827 """

5828 User Prompt:  
 5829

5830 Environment code examples:  
 5831 {ENV\_CODES\_EXAMPLE}

```

5832
5833 Learned tasks and environment code:
5834 {ENV_CODES_LEARNED}
5835
5836 Failed tasks and environment code:
5837 {ENV_CODES_FAILED}

```

## L.2 ENVIRONMENT GENERATOR

### System Prompt:

You are an expert in Python programming and reinforcement learning. Your goal is to implement an environment in PyBullet specifically designed to train a robot for a given task. You will be provided with the task description and with pairs of task description and environment code. Your objective is to write environment code that rigorously aligns with the task description, helping the robot learn the task as effectively as possible.

#### Instructions:

- Write code without using placeholders.
- Don't change the import statements.
- For each object, always define its size first, and ensure the object's initial position is set relative to the platform it starts on or any other object, as demonstrated in the provided environment code examples. For example, if an object is initialized on the ground, define its position as: [self.platform\_position[0], self.platform\_position[1], self.platform\_position[2] + self.platform\_size[2] / 2 + self.object\_size[2] / 2].
- Ensure the robot's initial position is set relative to the platform it starts on, as demonstrated in the provided environment code examples. For example, if the robot starts on a platform, its initial position should be set to [self.platform\_position[0], self.platform\_position[1], self.platform\_position[2] + self.platform\_size[2] / 2 + self.robot.links["base"].position\_init[2]].
- If the task involves navigating a terrain with obstacles, make sure that the robot cannot go around the obstacles.
- Implement the methods 'Env.reset()', 'Env.step()', 'Env.get\_task\_rewards()', 'Env.get\_terminated()', 'Env.get\_success()'. You can implement additional methods if needed.
- 'Env.get\_task\_rewards()' returns a dictionary with the different reward components to help the robot learn the task. You should implement dense reward components that are easy to optimize and defined in the range -10. to 10.
- 'Env.get\_terminated()' returns a boolean that indicates whether the episode is terminated.
- 'Env.get\_success()' returns a boolean that indicates whether the task is successfully completed.

Robot description:  
{ROBOT\_DESC}

Desired format:  
Environment code:  
'''python  
<environment code>  
'''

### User Prompt:

Pairs of task description and environment code:  
{ENV\_CODES\_EXAMPLE}

5886 Task description:  
 5887 {TASK\_DESC}  
 5888  
 5889

### L.3 ENVIRONMENT GENERATOR REFLECTION

#### System Prompt:

5893 You are an expert in Python programming and reinforcement learning. Your  
 5894 goal is to implement an environment in PyBullet specifically designed  
 5895 to train a robot for a given task. You will be provided with  
 5896 environment code examples, with an environment code that returns an  
 5897 error when executed and with the specific error that was encountered.  
 5898 Your objective is to reason about the error and provide a new,  
 5899 improved environment code with no error.

#### Instructions:

- 5900 – Write code without using placeholders.
- 5901 – Don't change the import statements.
- 5902 – For each object, always define its size first, and ensure the object's  
 5903 initial position is set relative to the platform it starts on or any  
 5904 other object, as demonstrated in the provided environment code  
 5905 examples. For example, if an object is initialized on the ground,  
 5906 define its position as: [self.platform\_position[0], self.  
 5907 platform\_position[1], self.platform\_position[2] + self.platform\_size  
 5908 [2] / 2 + self.object\_size[2] / 2].
- 5909 – Ensure the robot's initial position is set relative to the platform it  
 5910 starts on, as demonstrated in the provided environment code examples.  
 5911 For example, if the robot starts on a platform, its initial position  
 5912 should be set to [self.platform\_position[0], self.platform\_position  
 5913 [1], self.platform\_position[2] + self.platform\_size[2] / 2 + self.  
 5914 robot.links["base"].position\_init[2]].
- 5915 – If the task involves navigating a terrain with obstacles, make sure  
 5916 that the robot cannot go around the obstacles.
- 5917 – Implement the methods 'Env.reset()', 'Env.step()', 'Env.  
 5918 get\_task\_rewards()', 'Env.get\_terminated()', 'Env.get\_success()'. You  
 5919 can implement additional methods if needed.
- 5920 – 'Env.get\_task\_rewards()' returns a dictionary with the different reward  
 5921 components to help the robot learn the task. You should implement  
 5922 dense reward components that are easy to optimize and defined in the  
 5923 range -10. to 10.
- 5924 – 'Env.get\_terminated()' returns a boolean that indicates whether the  
 5925 episode is terminated.
- 5926 – 'Env.get\_success()' returns a boolean that indicates whether the task  
 5927 is successfully completed.

5928 Robot description:  
 5929 {ROBOT\_DESC}

5930 Desired format:

5931 How to solve the error:  
 5932 <reasoning>

5933 Environment code:  
 5934 '''python  
 5935 <environment code>  
 5936'''

#### User Prompt:

5937 Environment code examples:  
 5938 {ENV\_CODES\_EXAMPLE}

5939 Environment code with error:

5940 {ENV\_CODE}  
 5941  
 5942 Error:  
 5943 {ERROR}

5944

5945

## 5946 L.4 POST-GENERATION MODEL OF INTERESTINGNESS

5947

## 5948 System Prompt:

5949

5950

5951

5952

5953

5954

5955

5956

5957

5958

5959

5960

5961

5962

5963

5964

5965

5966

5967

5968

5969

5970

5971

5972

5973

5974

5975

5976

5977

5978

5979

5980

5981

5982

5983

5984

5985

5986

5987

5988

5989

5990

5991

5992

5993

You are an expert in curriculum learning and reinforcement learning. Your goal is to help a robot master a diverse set of interesting tasks in simulation using PyBullet. You will be provided with a list of old tasks and with a new task. Your objective is to determine whether the new task is interesting or not.

The new task can be considered interesting if one of the following is true, the new task is:

- Novel compared to the old tasks, to build a diverse skill set.
- Creative or surprising.
- Fun or engaging to watch.
- Not too easy for the robot to learn given its current skill set, progressing toward more complex challenges.
- Useful according to humans, making it worth learning.

Robot description:

{ROBOT\_DESC}

Desired format:

Reasoning for why the new task is interesting or not:

<reasoning>

Is the new task interesting?:

<Yes/No>

**User Prompt:**

Old tasks:

{ENV\_CODES\_EXAMPLE}

New task:

{ENV\_CODE}

5994 M FEW-SHOT EXAMPLES  
5995  
5996

```

5997 import numpy as np
5998 from oped.envs.r2d2.base import R2D2Env
5999
6000 class Env(R2D2Env):
6001     """
6002         Cross a pride-colored bridge to reach a platform.
6003
6004     Description:
6005         - A start platform and an end platform (each 3 m in size and 0.5 m in
6006             thickness) are placed 30 m apart.
6007         - The two platforms are connected by a bridge (2 m wide) divided in
6008             multiple segments. Each segment has a different color
6009             corresponding to the pride colors.
6010         The robot is initialized on the start platform.
6011         The task of the robot is to cross the bridge to reach the end
6012             platform as fast as possible.
6013
6014     Success:
6015         The task is successfully completed when the robot reaches the end
6016             platform.
6017
6018     Rewards:
6019         To help the robot complete the task:
6020             - The robot receives a reward for each time step it remains on the
6021                 bridge or platforms, encouraging steady progress.
6022             - The robot is rewarded based on how much it reduces the distance to
6023                 the end platform, incentivizing swift movement towards the goal.
6024
6025     Termination:
6026         The task terminates immediately if the robot falls off the start
6027             platform, any segment of the bridge, or the end platform.
6028     """
6029
6030     def __init__(self):
6031         super().__init__()
6032
6033         # Init start platform
6034         self.platform_size = [3., 3., 0.5]
6035         self.platform_start_position = [0., 0., 0.]
6036         self.platform_end_position = [self.platform_start_position[0] +
6037             30., self.platform_start_position[1], self.
6038             platform_start_position[2]]
6039         self.platform_start_id = self.create_box(mass=0., half_extents=[self.
6040             platform_size[0] / 2, self.platform_size[1] / 2, self.
6041             platform_size[2] / 2], position=self.platform_start_position,
6042             color=[0.8, 0.8, 0.8, 1.])
6043         self.platform_end_id = self.create_box(mass=0., half_extents=[self.
6044             platform_size[0] / 2, self.platform_size[1] / 2, self.
6045             platform_size[2] / 2], position=self.platform_end_position,
6046             color=[0.8, 0.8, 0.8, 1.])
6047
6048         # Init bridge
6049         self.bridge_length = self.platform_end_position[0] - self.
6050             platform_start_position[0] - self.platform_size[0]
6051         self.bridge_width = 2.
6052         pride_colors = [
6053             [1.0, 0.0, 0.0, 1.], # Red
6054             [1.0, 0.5, 0.0, 1.], # Orange
6055             [1.0, 1.0, 0.0, 1.], # Yellow
6056             [0.0, 0.5, 0.0, 1.], # Green
6057             [0.0, 0.0, 1.0, 1.], # Blue

```

```

6048     [0.7, 0.0, 1.0, 1.], # Violet
6049 ]
6050
6051     # Segment length
6052     num_colors = len(pride_colors)
6053     segment_size = self.bridge_length / num_colors
6054
6055     # Create segments
6056     for i, color in enumerate(pride_colors):
6057         segment_id = self.create_box(mass=0., half_extents=[
6058             segment_size / 2, self.bridge_width / 2, self.
6059             platform_size[2] / 2], position=[self.
6060             platform_start_position[0] + self.platform_size[0] / 2 +
6061             segment_size / 2 + i * segment_size, self.
6062             platform_start_position[1], self.platform_start_position
6063             [2]], color=color)
6064         self._p.changeDynamics(bodyUniqueId=segment_id, linkIndex=-1,
6065             lateralFriction=0.8, restitution=0.5)
6066
6067     def create_box(self, mass, half_extents, position, color):
6068         collision_shape_id = self._p.createCollisionShape(shapeType=self.
6069             _p.GEOM_BOX, halfExtents=half_extents)
6070         visual_shape_id = self._p.createVisualShape(shapeType=self._p.
6071             GEOM_BOX, halfExtents=half_extents, rgbaColor=color)
6072         return self._p.createMultiBody(baseMass=mass,
6073             baseCollisionShapeIndex=collision_shape_id,
6074             baseVisualShapeIndex=visual_shape_id, basePosition=position)
6075
6076     def get_object_position(self, object_id):
6077         return np.asarray(self._p.getBasePositionAndOrientation(object_id
6078             )[0])
6079
6080     def get_distance_to_object(self, object_id):
6081         object_position = self.get_object_position(object_id)
6082         robot_position = self.robot.links["base"].position
6083         return np.linalg.norm(object_position[:2] - robot_position[:2])
6084
6085     def reset(self):
6086         observation = super().reset()
6087
6088         # Reset robot position on start platform
6089         self._p.resetBasePositionAndOrientation(self.robot.robot_id, [
6090             self.platform_start_position[0], self.platform_start_position
6091             [1], self.platform_start_position[2] + self.platform_size[2]
6092             / 2 + self.robot.links["base"].position_init[2]], self.robot.
6093             links["base"].orientation_init)
6094
6095         return observation
6096
6097     def step(self, action):
6098         # Before taking action
6099         self.distance_to_platform_end = self.get_distance_to_object(self.
6100             platform_end_id)
6101
6102         observation, reward, terminated, truncated, info = super().step(
6103             action)
6104
6105         return observation, reward, terminated, truncated, info
6106
6107     def get_task_rewards(self, action):
6108         # After taking action
6109         new_distance_to_platform_end = self.get_distance_to_object(self.
6110             platform_end_id)
6111
6112         # Survival

```

```

6102     survival = 1.
6103
6104     # Reach end platform
6105     reach_platform_end = (self.distance_to_platform_end -
6106                             new_distance_to_platform_end) / self.dt
6107
6108     return {"survival": survival, "reach_platform_end":
6109             reach_platform_end}
6110
6111     def get_terminated(self, action):
6112         # Terminate if fall off
6113         return self.robot.links["base"].position[2] < self.
6114             platform_start_position[2]
6115
6116     def get_success(self):
6117         # Success if reach end platform
6118         is_on_platform_end = self.get_distance_to_object(self.
6119             platform_end_id) < self.platform_size[2] / 2
6120
6121         return is_on_platform_end
6122
6123
6124 import numpy as np
6125 from oped.envs.r2d2.base import R2D2Env
6126
6127
6128 class Env(R2D2Env):
6129     """
6130         Cross over lava on a boat to reach a target zone.
6131
6132     Description:
6133         - The lava is simulated with an orange, 10 x 10 m heightfield.
6134         - There are two platforms on either side of the lava, each measuring
6135             5 x 10 m. One serves as the start platform and the other as the
6136             end platform.
6137         - The boat is a box with dimensions 3 meters in length, 2 meters in
6138             width, and 0.2 meters in height. It is initialized next to the
6139             start platform at a random y-position.
6140         - The boat has a button that, when pressed, activates the boat to
6141             move over the lava at a speed of 3 meters per second.
6142         - The end platform has a target zone indicated by a green,
6143             transparent sphere.
6144         The robot's task is to jump onto the boat from the start platform,
6145         press the button to activate the boat, and travel across the lava
6146         to reach the end platform. The robot must then enter the target
6147         zone to complete the task.
6148
6149     Success:
6150         The task is successfully completed when the robot enters the target
6151         zone on the end platform.
6152
6153     Rewards:
6154         To guide the robot to complete the task:
6155             - The robot receives a reward for each time step it remains active
6156                 and does not fall off or touch the lava.
6157             - The robot is rewarded for making progress towards pressing the
6158                 button on the boat.
6159             - Additional rewards are given for progressing towards the target
6160                 zone, with a significant bonus for entering the target zone.
6161
6162     Termination:
6163         The task terminates immediately if the robot falls off the platform
6164             or the boat, or if it touches the simulated lava.
6165         """
6166
6167     def __init__(self):

```

```

6156     super().__init__()
6157
6158     # Init lava
6159     self.lava_size = [10., 10.]
6160     self.lava_height = 0.1
6161     self.lava_position = [0., 0., 0.]
6162     self.lava_id = self.create_heightfield(
6163         size=self.lava_size,
6164         height_max=self.lava_height, # create small bumps to create
6165         # a fluid-like surface
6166         position=self.lava_position,
6167         resolution=20, # number of points per meter
6168         repeats=2,
6169     )
6170     self._p.changeVisualShape(objectUniqueId=self.lava_id, linkIndex
6171     =-1, rgbaColor=[1., 0.3, 0.1, 1.]) # change to lava color
6172
6173     # Init platforms
6174     self.platform_size = [5., self.lava_size[1], 1.]
6175     self.platform_start_position = [self.lava_position[0] - self.
6176         lava_size[0] / 2 - self.platform_size[0] / 2, self.
6177         lava_position[1], self.lava_position[2]]
6178     self.platform_end_position = [self.lava_position[0] + self.
6179         lava_size[0] / 2 + self.platform_size[0] / 2, self.
6180         lava_position[1], self.lava_position[2]]
6181     self.platform_start_id = self.create_box(mass=0., half_extents=[
6182         self.platform_size[0] / 2, self.platform_size[1] / 2, self.
6183         platform_size[2] / 2], position=self.platform_start_position,
6184         color=[0.3, 0.3, 0.3, 1.])
6185     self.platform_end_id = self.create_box(mass=0., half_extents=[[
6186         self.platform_size[0] / 2, self.platform_size[1] / 2, self.
6187         platform_size[2] / 2], position=self.platform_end_position,
6188         color=[0.3, 0.3, 0.3, 1.]])
6189     self._p.changeDynamics(bodyUniqueId=self.platform_start_id,
6190     linkIndex=-1, lateralFriction=0.8, restitution=0.5)
6191     self._p.changeDynamics(bodyUniqueId=self.platform_end_id,
6192     linkIndex=-1, lateralFriction=0.8, restitution=0.5)
6193
6194     # Init boat
6195     self.boat_size = [3., 2., 0.2]
6196     self.boat_position_init = [self.lava_position[0] - self.lava_size
6197         [0] / 2 + self.boat_size[0] / 2, self.lava_position[1], self.
6198         boat_size[2] / 2]
6199     self.boat_speed = 3.
6200     self.boat_id = self.create_box(mass=0., half_extents=[self.
6201         boat_size[0] / 2, self.boat_size[1] / 2, self.boat_size[2] /
6202         2], position=self.boat_position_init, color=[0.8, 0.8, 0.8,
6203         1.])
6204     self._p.changeDynamics(bodyUniqueId=self.boat_id, linkIndex=-1,
6205     lateralFriction=0.8, restitution=0.5)
6206
6207     # Init button
6208     self.button_radius = 0.25
6209     self.button_height = 0.25
6210     self.button_position_init = [self.boat_position_init[0] + self.
6211         boat_size[0] / 4, self.lava_position[1], self.
6212         boat_position_init[2] + self.boat_size[2] / 2 + self.
6213         button_height / 2] # put button on the right side of the
6214         boat
6215     self.button_id = self.create_cylinder(mass=0., radius=self.
6216         button_radius, height=self.button_height, position=self.
6217         button_position_init, color=[0., 0.5, 0., 1.])
6218
6219     # Init target zone
6220     self.target_zone_radius = 1.5

```

```

6210         self.target_zone_id = self.create_sphere(mass=0., radius=self.
6211             target_zone_radius, collision=False, position=[self.
6212                 platform_end_position[0], self.platform_end_position[1], self.
6213                     .platform_end_position[2] + self.platform_size[2] / 2], color
6214                         =[0., 1., 0., 0.5])
6215
6216         self.objects_on_boat = [self.button_id]
6217
6218     def create_box(self, mass, half_extents, position, color):
6219         collision_shape_id = self._p.createCollisionShape(shapeType=self._p.GEOM_BOX, halfExtents=half_extents)
6220         visual_shape_id = self._p.createVisualShape(shapeType=self._p.GEOM_BOX, halfExtents=half_extents, rgbaColor=color)
6221         return self._p.createMultiBody(baseMass=mass,
6222             baseCollisionShapeIndex=collision_shape_id,
6223                 baseVisualShapeIndex=visual_shape_id, basePosition=position)
6224
6225     def create_cylinder(self, mass, radius, height, position, color):
6226         collision_shape_id = self._p.createCollisionShape(shapeType=self._p.GEOM_CYLINDER, radius=radius, height=height)
6227         visual_shape_id = self._p.createVisualShape(shapeType=self._p.GEOM_CYLINDER, radius=radius, length=height, rgbaColor=color)
6228         return self._p.createMultiBody(baseMass=mass,
6229             baseCollisionShapeIndex=collision_shape_id,
6230                 baseVisualShapeIndex=visual_shape_id, basePosition=position)
6231
6232     def create_sphere(self, mass, radius, collision, position, color):
6233         if collision:
6234             collision_shape_id = self._p.createCollisionShape(shapeType=self._p.GEOM_SPHERE, radius=radius)
6235             visual_shape_id = self._p.createVisualShape(shapeType=self._p.GEOM_SPHERE, radius=radius, rgbaColor=color)
6236             return self._p.createMultiBody(baseMass=mass,
6237                 baseCollisionShapeIndex=collision_shape_id,
6238                     baseVisualShapeIndex=visual_shape_id, basePosition=
6239                         position)
6240         else:
6241             visual_shape_id = self._p.createVisualShape(shapeType=self._p.GEOM_SPHERE, radius=radius, rgbaColor=color)
6242             return self._p.createMultiBody(baseMass=mass,
6243                 baseVisualShapeIndex=visual_shape_id, basePosition=
6244                     position)
6245
6246     def create_heightfield(self, size, height_max, position, resolution,
6247         repeats=2):
6248         heightfield_data = np.random.uniform(low=0., high=height_max,
6249             size=(int(size[0]) * resolution / repeats), int(size[1] *
6250                 resolution / repeats)))
6251         heightfield_data = np.repeat(np.repeat(heightfield_data, repeats,
6252             axis=0), repeats, axis=1)
6253         mesh_scale = [1/resolution, 1/resolution, 1.]
6254         heightfield_collision_shape_id = self._p.createCollisionShape(
6255             shapeType=self._p.GEOM_HEIGHTFIELD,
6256                 meshScale=mesh_scale,
6257                     heightfieldData=heightfield_data.reshape(-1),
6258                         numHeightfieldRows=heightfield_data.shape[0],
6259                             numHeightfieldColumns=heightfield_data.shape[1],
6260                         )
6261         return self._p.createMultiBody(baseMass=0.,
6262             baseCollisionShapeIndex=heightfield_collision_shape_id,
6263                 basePosition=[position[0], position[1], position[2] +
6264                     mesh_scale[2] * height_max / 2])
6265
6266     def get_object_position(self, object_id):

```

```

6264     return np.asarray(self._p.getBasePositionAndOrientation(object_id
6265         )[0])
6266
6267     def get_distance_to_object(self, object_id):
6268         object_position = self.get_object_position(object_id)
6269         robot_position = self.robot.links["base"].position
6270         return np.linalg.norm(object_position[:2] - robot_position[:2])
6271
6272     def reset(self):
6273         observation = super().reset()
6274
# Reset boat position
6275         boat_y_init = np.random.uniform(low=-self.lava_size[1] / 2 + self.
6276             .boat_size[1] / 2, high=self.lava_size[1] / 2 - self.
6277             .boat_size[1] / 2) # randomize y position
6278         self._p.resetBasePositionAndOrientation(self.boat_id, [self.
6279             .boat_position_init[0], boat_y_init, self.boat_position_init
6280             [2]], [0., 0., 0., 1.])
6281
# Reset button position
6282         self._p.resetBasePositionAndOrientation(self.button_id, [self.
6283             .button_position_init[0], boat_y_init, self.
6284             .button_position_init[2]], [0., 0., 0., 1.])
6285
# Reset target zone
6286         target_zone_y = np.random.uniform(low=-self.lava_size[1] / 2 +
6287             self.target_zone_radius, high=self.lava_size[1] / 2 - self.
6288             .target_zone_radius) # randomize y position
6289         self.target_zone_position = [self.platform_end_position[0],
6290             target_zone_y, self.platform_end_position[2] + self.
6291             .platform_size[2] / 2]
6292         self._p.resetBasePositionAndOrientation(self.target_zone_id, self
6293             .target_zone_position, [0., 0., 0., 1.])
6294
# Reset robot position
6295         self._p.resetBasePositionAndOrientation(self.robot.robot_id, [
6296             self.platform_start_position[0], self.platform_start_position
6297             [1], self.platform_start_position[2] + self.platform_size[2]
6298             / 2 + self.robot.links["base"].position_init[2]], self.robot.
6299             links["base"].orientation_init)
6300
6301     return observation
6302
6303     def step(self, action):
# Before taking action
6304         self.distance_to_button = self.get_distance_to_object(self.
6305             .button_id)
6306         self.distance_to_target_zone = self.get_distance_to_object(self.
6307             .target_zone_id)
6308         self.has_touched_platform_end = len(self._p.getContactPoints(
6309             bodyA=self.robot.robot_id, bodyB=self.platform_end_id)) > 0
6310
6311         observation, reward, terminated, truncated, info = super().step(
6312             action)
6313
# Check if button is pressed
6314         contact_points = self._p.getContactPoints(bodyA=self.robot.
6315             robot_id, bodyB=self.button_id)
6316         button_pressed = len(contact_points) > 0
6317
if button_pressed:
# Move boat and everything on boat forward
    for body_id in [self.boat_id] + self.objects_on_boat:
        body_position = self.get_object_position(body_id)

```

```

6318         new_object_position = body_position + np.array([self.
6319             boat_speed * self.dt, 0., 0.])
6320         self._p.resetBasePositionAndOrientation(body_id,
6321             new_object_position, [0., 0., 0., 1.])
6322
6323     return observation, reward, terminated, truncated, info
6324
6325     def get_task_rewards(self, action):
6326         # After taking action
6327         new_distance_to_button = self.get_distance_to_object(self.
6328             button_id)
6329         new_distance_to_target_zone = self.get_distance_to_object(self.
6330             target_zone_id)
6331
6332         # Survival
6333         survival = 1.
6334
6335         # Reach button
6336         reach_button = (self.distance_to_button - new_distance_to_button) /
6337             self.dt
6338
6339         # Reach target zone
6340         reach_target_zone = (self.distance_to_target_zone -
6341             new_distance_to_target_zone) / self.dt
6342         if self.distance_to_target_zone < self.target_zone_radius:
6343             reach_target_zone += 5.
6344
6345         return {"survival": survival, "reach_button": reach_button, "reach_target_zone": reach_target_zone}
6346
6347     def get_terminated(self, action):
6348         # Terminate if touch lava
6349         contact_points = self._p.getContactPoints(bodyA=self.robot.
6350             robot_id, bodyB=self.lava_id)
6351         is_touching_lava = len(contact_points) > 0
6352
6353         # Terminate if fall off
6354         is_fall_off = self.robot.links["base"].position[2] < self.
6355             platform_start_position[2]
6356         return is_touching_lava or is_fall_off
6357
6358     def get_success(self):
6359         # Success if stand in the target zone
6360         distance_to_target_zone = self.get_distance_to_object(self.
6361             target_zone_id)
6362         return distance_to_target_zone < self.target_zone_radius
6363
6364
6365     import numpy as np
6366     from oped.envs.r2d2.base import R2D2Env
6367
6368     class Env(R2D2Env):
6369         """
6370             Descend a series of stairs to reach the ground.
6371
6372             Description:
6373             - The environment consists of a ground platform (1000 m x 10 m x 10 m
6374                 ) and a set of 10 steps.
6375             - Each step has dimensions of 1 m in length, 10 m in width, and 0.2 m
6376                 in height.
6377             - The steps are positioned to form a descending staircase starting
6378                 from an initial height, with each subsequent step lower than the
6379                 previous one.
6380             The robot is initialized at the top of the stairs.

```

```

6372
6373 Success:
6374 The task is completed when the robot successfully descends the stairs
6375 and touches the ground platform.
6376
6377 Rewards:
6378 The help the robot complete the task:
6379 - The robot is rewarded for survival at each time step.
6380 - The robot is rewarded for forward velocity, incentivizing it to
6381 move down the stairs.
6382
6383 Termination:
6384 The task terminates immediately if the robot falls off the stairs or
6385 the ground platform.
6386 """
6387
6388 def __init__(self):
6389     super().__init__()
6390
6391     # Init ground
6392     self.ground_size = [1000., 10., 10.]
6393     self.ground_position = [0., 0., 0.]
6394     self.ground_id = self.create_box(mass=0., half_extents=[self.
6395         ground_size[0] / 2, self.ground_size[1] / 2, self.ground_size
6396         [2] / 2], position=self.ground_position, color=[0.5, 0.5,
6397         0.5, 1.])
6398     self._p.changeDynamics(bodyUniqueId=self.ground_id, linkIndex=-1,
6399     lateralFriction=0.8, restitution=0.5)
6400
6401     # Init stairs
6402     self.num_steps = 10
6403     self.step_size = [1.0, 10., 0.2]
6404     self.step_position_init = [self.ground_position[0], self.
6405         ground_position[1], self.ground_position[2] + self.
6406         ground_size[2] / 2 + self.num_steps * self.step_size[2]]
6407     self.create_stairs_down(step_size=self.step_size,
6408     step_position_init=self.step_position_init, num_steps=self.
6409     num_steps)
6410
6411 def create_box(self, mass, half_extents, position, color):
6412     collision_shape_id = self._p.createCollisionShape(shapeType=self.
6413         _p.GEOM_BOX, halfExtents=half_extents)
6414     visual_shape_id = self._p.createVisualShape(shapeType=self._p.
6415         GEOM_BOX, halfExtents=half_extents, rgbaColor=color)
6416     return self._p.createMultiBody(baseMass=mass,
6417     baseCollisionShapeIndex=collision_shape_id,
6418     baseVisualShapeIndex=visual_shape_id, basePosition=position)
6419
6420 def create_stairs_down(self, step_size, step_position_init, num_steps
6421 )::
6422     color_1 = np.array([1., 0., 0.])
6423     color_2 = np.array([0., 0., 1.])
6424     for i in range(num_steps):
6425         step_position = [step_position_init[0] + i * step_size[0],
6426             step_position_init[1], step_position_init[2] - i *
6427             step_size[2]]
6428         interpolation = i / (num_steps - 1)
6429         step_color = (1 - interpolation) * color_1 + interpolation *
6430             color_2 # shade steps for visualization
6431         self.create_box(mass=0., half_extents=[step_size[0] / 2,
6432             step_size[1] / 2, step_size[2] / 2], position=
6433             step_position, color=np.append(step_color, 1.))
6434
6435 def reset(self):
6436     observation = super().reset()

```

```

6426
6427     # Reset robot position at the top of the stairs
6428     self._p.resetBasePositionAndOrientation(self.robot.robot_id, [
6429         self.step_position_init[0], self.step_position_init[1], self.
6430         step_position_init[2] + self.step_size[2] / 2 + self.robot.
6431         links["base"].position_init[2]], self.robot.links["base"].
6432         orientation_init)
6433
6434     return observation
6435
6436     def step(self, action):
6437         # Before taking action
6438         self.position = self.robot.links["base"].position
6439
6440         observation, reward, terminated, truncated, info = super().step(
6441             action)
6442
6443         return observation, reward, terminated, truncated, info
6444
6445     def get_task_rewards(self, action):
6446         # After taking action
6447         new_position = self.robot.links["base"].position
6448
6449         # Survival
6450         survival = 1.
6451
6452         # Forward velocity
6453         forward_velocity = (new_position[0] - self.position[0]) / self.dt
6454
6455         return {"survival": survival, "forward_velocity":
6456             forward_velocity}
6457
6458     def get_terminated(self, action):
6459         # Terminate if fall off
6460         return self.robot.links["base"].position[2] < self.
6461             ground_position[2]
6462
6463
6464     def get_success(self):
6465         # Success if reach end stairs and touch ground
6466         contact_points = self._p.getContactPoints(bodyA=self.robot.
6467             robot_id, bodyB=self.ground_id)
6468         is_on_ground = len(contact_points) > 0
6469         return is_on_ground
6470
6471
6472     import numpy as np
6473     from oped.envs.r2d2.base import R2D2Env
6474
6475
6476     class Env(R2D2Env):
6477         """
6478             Activate a lever to open a door and move through the door.
6479
6480             Description:
6481             - The environment consists of a large platform measuring 1000 x 10 x
6482                 0.1 meters.
6483             - The robot is initialized at a fixed position on the platform.
6484             - A door with dimensions 0.5 x 2 x 2 meters is positioned on the
6485                 platform, 5 m away from the robot, initially closed.
6486             - The door is flanked by walls to prevent the robot from bypassing it
6487             .
6488             - A lever is placed on the platform, 2 meters to the left of the door
6489             .
6490             - The task of the robot is to move to the lever, activate it to open
6491                 the door, and then pass through the door.
6492
6493
6494
6495
6496
6497
6498
6499
6500
6501
6502
6503
6504
6505
6506
6507
6508
6509
6510
6511
6512
6513
6514
6515
6516
6517
6518
6519
6520
6521
6522
6523
6524
6525
6526
6527
6528
6529
6530
6531
6532
6533
6534
6535
6536
6537
6538
6539
6540
6541
6542
6543
6544
6545
6546
6547
6548
6549
6550
6551
6552
6553
6554
6555
6556
6557
6558
6559
6560
6561
6562
6563
6564
6565
6566
6567
6568
6569
6570
6571
6572
6573
6574
6575
6576
6577
6578
6579
6580
6581
6582
6583
6584
6585
6586
6587
6588
6589
6590
6591
6592
6593
6594
6595
6596
6597
6598
6599
6600
6601
6602
6603
6604
6605
6606
6607
6608
6609
6610
6611
6612
6613
6614
6615
6616
6617
6618
6619
6620
6621
6622
6623
6624
6625
6626
6627
6628
6629
6630
6631
6632
6633
6634
6635
6636
6637
6638
6639
6640
6641
6642
6643
6644
6645
6646
6647
6648
6649
6650
6651
6652
6653
6654
6655
6656
6657
6658
6659
6660
6661
6662
6663
6664
6665
6666
6667
6668
6669
6670
6671
6672
6673
6674
6675
6676
6677
6678
6679
6680
6681
6682
6683
6684
6685
6686
6687
6688
6689
6690
6691
6692
6693
6694
6695
6696
6697
6698
6699
6700
6701
6702
6703
6704
6705
6706
6707
6708
6709
6710
6711
6712
6713
6714
6715
6716
6717
6718
6719
6720
6721
6722
6723
6724
6725
6726
6727
6728
6729
6730
6731
6732
6733
6734
6735
6736
6737
6738
6739
6740
6741
6742
6743
6744
6745
6746
6747
6748
6749
6750
6751
6752
6753
6754
6755
6756
6757
6758
6759
6760
6761
6762
6763
6764
6765
6766
6767
6768
6769
6770
6771
6772
6773
6774
6775
6776
6777
6778
6779
6780
6781
6782
6783
6784
6785
6786
6787
6788
6789
6790
6791
6792
6793
6794
6795
6796
6797
6798
6799
6800
6801
6802
6803
6804
6805
6806
6807
6808
6809
6810
6811
6812
6813
6814
6815
6816
6817
6818
6819
6820
6821
6822
6823
6824
6825
6826
6827
6828
6829
6830
6831
6832
6833
6834
6835
6836
6837
6838
6839
6840
6841
6842
6843
6844
6845
6846
6847
6848
6849
6850
6851
6852
6853
6854
6855
6856
6857
6858
6859
6860
6861
6862
6863
6864
6865
6866
6867
6868
6869
6870
6871
6872
6873
6874
6875
6876
6877
6878
6879
6880
6881
6882
6883
6884
6885
6886
6887
6888
6889
6890
6891
6892
6893
6894
6895
6896
6897
6898
6899
6899

```

```

6480
6481     Success:
6482     The task is successfully completed if the robot passes through the
6483     door and moves more than 10 m beyond the initial position.
6484
6485     Rewards:
6486     To guide the robot to complete the task:
6487     - The robot receives a survival reward at each time step.
6488     - The robot is rewarded for decreasing its distance to the lever.
6489     - The robot receives a bonus rewards for activating the lever to open
6490       the door.
6491     - Once the door is open, the robot is rewarded for moving forward.
6492
6493     Termination:
6494     The task terminates immediately if the robot falls off the stairs or
6495     the ground platform.
6496     """
6497
6498     def __init__(self):
6499         super().__init__()
6500
6501         self.robot_position_init = [0., 0., 0.]
6502
6503         # Init platform
6504         self.platform_size = [1000., 10., 0.1]
6505         self.platform_position = [self.robot_position_init[0] + self.
6506           platform_size[0] / 2 - 2., self.robot_position_init[1], self.
6507           robot_position_init[2] - self.platform_size[2] / 2] # offset
6508           by 2 m to avoid off-edge or on-edge placement
6509         self.platform_id = self.create_box(mass=0., half_extents=[self.
6510           platform_size[0] / 2, self.platform_size[1] / 2, self.
6511           platform_size[2] / 2], position=self.platform_position, color
6512           =[0.5, 0.5, 0.5, 1.])
6513         self._p.changeDynamics(bodyUniqueId=self.platform_id, linkIndex
6514           =-1, lateralFriction=0.8, restitution=0.5)
6515
6516         # Init door
6517         self.door_size = [0.5, 2., 2.]
6518         self.door_position_init = [self.robot_position_init[0] + 5., self.
6519           .platform_position[1], self.platform_position[2] + self.
6520           platform_size[2] / 2 + self.door_size[2] / 2]
6521         self.door_id = self.create_box(mass=0., half_extents=[self.
6522           door_size[0] / 2, self.door_size[1] / 2, self.door_size[2] /
6523           2], position=self.door_position_init, color=[1., 0., 0., 1.])
6524         self.door_open = False
6525
6526         # Init wall
6527         self.wall_size = [self.door_size[0], (self.platform_size[1] -
6528           self.door_size[1]) / 2, self.door_size[2]] # walls plus door
6529           span the full platform to prevent robot to go around
6530         self.create_box(mass=0., half_extents=[self.wall_size[0] / 2,
6531           self.wall_size[1] / 2, self.wall_size[2] / 2], position=[self.
6532           door_position_init[0], self.door_position_init[1] + self.
6533           door_size[1] / 2 + self.wall_size[1] / 2, self.
6534           platform_position[2] + self.platform_size[2] / 2 + self.
6535           wall_size[2] / 2], color=[0., 0., 1., 1.]) # left section
6536         self.create_box(mass=0., half_extents=[self.wall_size[0] / 2,
6537           self.wall_size[1] / 2, self.wall_size[2] / 2], position=[self.
6538           door_position_init[0], self.door_position_init[1] - self.
6539           door_size[1] / 2 - self.wall_size[1] / 2, self.
6540           platform_position[2] + self.platform_size[2] / 2 + self.
6541           wall_size[2] / 2], color=[0., 0., 1., 1.]) # right section
6542
6543         # Init lever
6544         self.lever_radius = 0.05

```

```

6534     self.lever_height = 0.5
6535     lever_position = [self.door_position_init[0] - 2., self.door_size
6536         [1], self.platform_position[2] + self.platform_size[2] / 2 +
6537             self.lever_height / 2] # two meters to the left of the door
6538             on the platform
6539     self.lever_id = self.create_cylinder(mass=0., radius=self.
6540         lever_radius, height=self.lever_height, position=
6541         lever_position, color=[0.5, 0.25, 0., 1.])
6542
6543     def create_box(self, mass, half_extents, position, color):
6544         collision_shape_id = self._p.createCollisionShape(shapeType=self._p.GEOM_BOX, halfExtents=half_extents)
6545         visual_shape_id = self._p.createVisualShape(shapeType=self._p.GEOM_BOX, halfExtents=half_extents, rgbaColor=color)
6546         return self._p.createMultiBody(baseMass=mass,
6547             baseCollisionShapeIndex=collision_shape_id,
6548             baseVisualShapeIndex=visual_shape_id, basePosition=position)
6549
6550     def create_cylinder(self, mass, radius, height, position, color):
6551         collision_shape_id = self._p.createCollisionShape(shapeType=self._p.GEOM_CYLINDER, radius=radius, height=height)
6552         visual_shape_id = self._p.createVisualShape(shapeType=self._p.GEOM_CYLINDER, radius=radius, length=height, rgbaColor=color)
6553         return self._p.createMultiBody(baseMass=mass,
6554             baseCollisionShapeIndex=collision_shape_id,
6555             baseVisualShapeIndex=visual_shape_id, basePosition=position)
6556
6557     def get_object_position(self, object_id):
6558         return np.asarray(self._p.getBasePositionAndOrientation(object_id)[0])
6559
6560     def get_distance_to_object(self, object_id):
6561         object_position = self.get_object_position(object_id)
6562         robot_position = self.robot.links["base"].position
6563         return np.linalg.norm(object_position[:2] - robot_position[:2])
6564
6565     def reset(self):
6566         observation = super().reset()
6567
6568         # Reset door
6569         self.door_open = False
6570         self._p.resetBasePositionAndOrientation(self.door_id, self.door_position_init, [0., 0., 0., 1.])
6571
6572         # Reset robot position
6573         self._p.resetBasePositionAndOrientation(self.robot.robot_id, [
6574             self.robot_position_init[0], self.robot_position_init[1],
6575             self.robot_position_init[2] + self.robot.links["base"].position_init[2]], self.robot.links["base"].orientation_init)
6576
6577         return observation
6578
6579     def step(self, action):
6580         # Before taking action
6581         self.position = self.robot.links["base"].position
6582         self.distance_to_lever = self.get_distance_to_object(self.lever_id)
6583
6584         observation, reward, terminated, truncated, info = super().step(action)
6585
6586         contact_points = self._p.getContactPoints(bodyA=self.robot.robot_id, bodyB=self.lever_id)
6587         if len(contact_points) > 0 and not self.door_open:
6588             self.door_open = True

```

```

6588         self._p.resetBasePositionAndOrientation(self.door_id, [self.
6589             door_position_init[0], self.door_position_init[1] + self.
6590             door_size[1], self.door_position_init[2]], [0., 0., 0.,
6591             1.])
6592
6593     return observation, reward, terminated, truncated, info
6594
6595     def get_task_rewards(self, action):
6596         # After taking action
6597         new_position = self.robot.links["base"].position
6598         new_distance_to_lever = self.get_distance_to_object(self.lever_id
6599             )
6600
6601         # Survival
6602         survival = 1.
6603
6604         # Reach lever
6605         if not self.door_open and len(self._p.getContactPoints(bodyA=self.
6606             .robot.robot_id, bodyB=self.lever_id)) == 0:
6607             reach_lever = (self.distance_to_lever - new_distance_to_lever
6608                 ) / self.dt
6609         elif not self.door_open and len(self._p.getContactPoints(bodyA=
6610             self.robot.robot_id, bodyB=self.lever_id)) > 0:
6611             reach_lever = 10.
6612         else:
6613             reach_lever = 0.
6614
6615         # Forward velocity
6616         if self.door_open:
6617             forward_velocity = (new_position[0] - self.position[0]) /
6618                 self.dt
6619         else:
6620             forward_velocity = 0.
6621
6622         return {"survival": survival, "reach_lever": reach_lever, "
6623             "forward_velocity": forward_velocity}
6624
6625     def get_terminated(self, action):
6626         # Terminate if fall off
6627         return self.robot.links["base"].position[2] < self.
6628             platform_position[2]
6629
6630     def get_success(self):
6631         # Success if pass through door
6632         return self.robot.links["base"].position[0] > 10.
6633
6634
6635
6636
6637
6638
6639
6640
6641

```

6642 **N TASK DESCRIPTION SEEDS**

6643

6644 Cross a pride-colored bridge with gaps to reach a platform.

6645

6646 **Description:**

- 6647 – A start platform and an end platform (each 3 m in size and 0.5 m in thickness) are placed 50 m apart.
- 6648 – The two platforms are connected by a bridge (2 m wide) divided in multiple segments. Each segment has a different color corresponding to the pride colors.
- 6649 – The segments are separated by gaps measuring 2 m.
- 6650 The robot is initialized on the start platform.
- 6651 The task of the robot is to cross the bridge to reach the end platform as fast as possible.

6652 **Success:**

6653 The task is successfully completed when the robot reaches the end platform.

6654

6655 **Rewards:**

6656 To help the robot complete the task:

- 6657 – The robot receives a reward for each time step it remains standing on the bridge or platforms, encouraging steady progress.
- 6658 – The robot is rewarded based on how much it reduces the distance to the end platform, incentivizing swift movement towards the goal.

6659 **Termination:**

6660 The task terminates immediately if the robot falls off the start platform , any segment of the bridge, or the end platform.

6661

6662 Ascend a series of stairs to reach a platform.

6663

6664 **Description:**

- 6665 – The environment consists of a ground platform (1000 m x 10 m x 10 m) and a set of 10 steps.
- 6666 – Each step has dimensions of 1 m in length, 10 m in width, and 0.2 m in height.
- 6667 – The steps are positioned to form an ascending staircase, with each subsequent step higher than the previous one.

6668 The robot is initialized on the ground at the bottom of the stairs.

6669

6670 **Success:**

6671 The task is completed when the robot successfully ascends the stairs and reaches the top platform.

6672

6673 **Rewards:**

6674 To help the robot complete the task:

- 6675 – The robot is rewarded for survival at each time step.
- 6676 – The robot is rewarded for forward velocity, incentivizing it to move up the stairs.

6677 **Termination:**

6678 The task terminates immediately if the robot falls off the stairs or the top platform.

6679

6680 Kick a ball into a goal.

6681

6682 **Description:**

- 6683 – The environment consists of a large flat ground measuring 1000 x 1000 x 10 meters.
- 6684 – A ball with a radius of 0.5 meters is placed randomly on the ground.
- 6685 – The goal is defined by two goal posts, each 2 meters high and placed 3 meters apart, forming a goal area.

6696     – The robot is initialized at a fixed position on the ground.  
6697     – The task of the robot is to move across the ground, reach the ball, and  
6698        kick it into the goal.

6699     **Success:**  
6700        The task is successfully completed if the robot kicks the ball so that it  
6701        passes between the two goal posts.

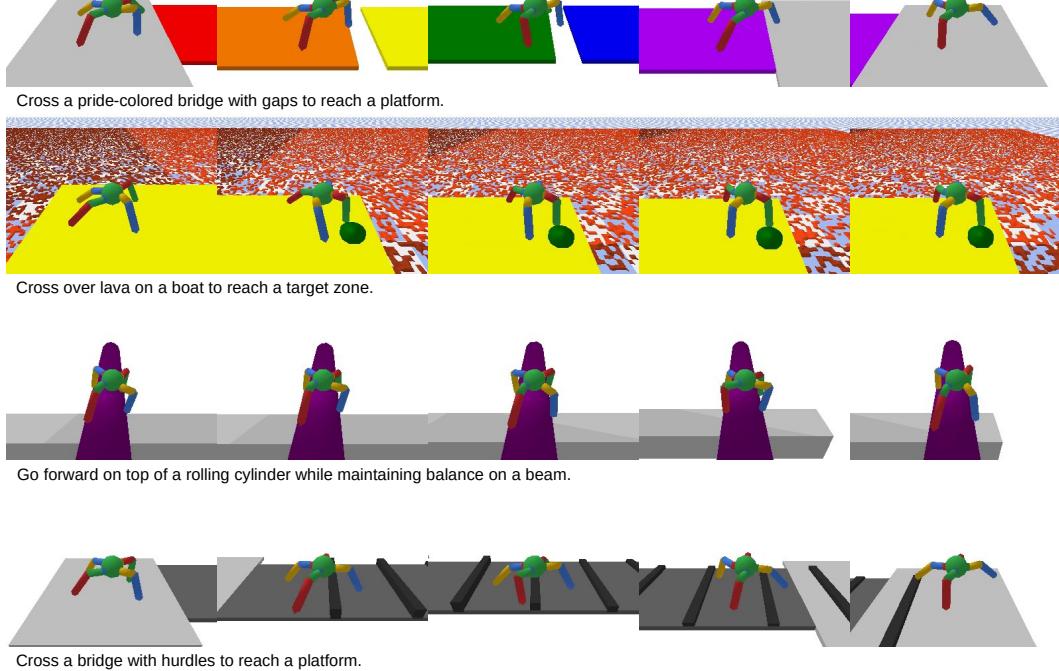
6702     **Rewards:**  
6703        To help the robot complete the task:  
6704        – The robot is rewarded for survival at each time step.  
6705        – The robot is rewarded for decreasing its distance to the ball.  
6706        – The robot is rewarded for kicking the ball towards the goal, with  
6707        additional rewards for successfully kicking the ball into the goal.

6708     **Termination:**  
6709        The task does not have a specific termination condition.

6711  
6712  
6713  
6714  
6715  
6716  
6717  
6718  
6719  
6720  
6721  
6722  
6723  
6724  
6725  
6726  
6727  
6728  
6729  
6730  
6731  
6732  
6733  
6734  
6735  
6736  
6737  
6738  
6739  
6740  
6741  
6742  
6743  
6744  
6745  
6746  
6747  
6748  
6749

## 6750 O EXPERIMENTS ON DIFFERENT ROBOTS AND ACTION SPACES

6752 OMNI-EPIC is designed to accommodate a wide range of robotic systems, regardless of the robot  
 6753 type or action space. To demonstrate the flexibility of our approach, we train an Ant robot on  
 6754 generated tasks using OMNI-EPIC. The Ant robot is a 3D quadruped consisting of a torso with free  
 6755 rotational movement and four legs, each composed of two segments (Towers et al., 2024). Apart  
 6756 from the robot type and action space, all settings, including the observation space and RL algorithm,  
 6757 are kept consistent with those used for the R2D2 robot. The Ant robot’s action space is defined as a  
 6758 continuous 8-dimensional vector, with each element bounded between -1 and 1.



**Figure 44: Ant robot successfully completing different generated tasks.** These examples highlight OMNI-EPIC’s ability to train various robot types and operate across different action spaces.