PDSim: Planning Domain Simulation and Animation with the Unity Game Engine

Emanuele De Pellegrin, Ronald P. A. Petrick

Edinburgh Centre for Robotics
Heriot-Watt University
Edinburgh, Scotland, United Kingdom
ed50@hw.ac.uk, R.Petrick@hw.ac.uk

Abstract

Modelling planning domains that are correct and robust can be a challenging problem, especially in real-world domains. This paper presents an overview of the current state of the Planning Domain Simulation (PDSim) project, an asset for the Unity game engine to simulate plans in a 2D or 3D environment with custom animations and graphics effects. PDSim aims to provide an intuitive tool for users to define animations without the need to learn a new scripting language, using the Unity game engine's internal and industry-standard visual scripting language in order to quickly evaluate the validity of planning models. PDSim fills an important gap in the area of planning simulation and validation: simulating a planning problem using 3D or 2D graphics and animation techniques can help the user to quickly evaluate the quality and correctness of a plan, and improve the design of a planning domain and problem. This paper presents the current state of PDSim development and future plans for the project.

Introduction

The task of modelling planning domains that are both correct and robust can be a challenging problem, especially in real-world domains. For instance, consider the following robot planning task: a set of robots are deployed in a factory to help with the warehouse logistics. The robots can navigate on a predefined grid map with simple 4-way movements, pick up and drop boxes, and deliver objects to a van parked in the warehouse. The problem also imposes a few limitations: the robots cannot cross each other, and the vans can only accept a specific box.

The above problem could be viewed as a slightly modified version of the sequential *Floor Tile* domain from the 2011 International Planning Competition (IPC):[1] a real-world inspired problem that can be modelled using a representation language such as PDDL (McDermott et al. 1998) and solved with classical automated planning techniques. The grid could be modelled as a set of interconnected nodes representing locations in the warehouse for objects and agents (e.g., vans, boxes, and robots), as illustrated in Figure 1. A trivial example of a goal might be to ensure that particular objects are in specific locations, e.g., *box1* is in *van1*. Using this domain model, we can quickly find a *valid* solution to the problem. For instance, Figure 2 (left) shows a plan generated by the FastDownward planner (Helmert 2006) for the problem in Figure 1, where a robot moves to grid cell *(1,0)* to pick up the box before delivering it to the van at *(0,0)*.

Figure 2 (right) shows an alternative action sequence, generated using an incorrect version of the domain. Although the plan is similar to the one on the left, it is *incorrect*: the robot executes the pickup action when in grid cell *(0,0)* before loading the van. (This plan is the result of a missing precondition on the pickup action which normally ensures that the robot and object are in the same cell.) While this kind of error can be trivial to debug and correct by an expert knowledge engineer, this isn't always the case for students and newcomers to languages such as PDDL. Catching modelling errors (i.e., incorrect logic in action preconditions and effects, missed predicates in an *init* block, etc.) can still be difficult due to the complexity of the knowledge that needs to be specified and the level of abstraction that is often required for ensuring the generation of tractable solutions.

In this paper, we present the current state of the Planning Domain Simulation (PDSim) (De Pellegrin 2020; De Pellegrin and Petrick 2021, 2022) system, a framework for visualising and simulating classical and temporal planning problems. PDDL is used to define the domain knowledge and the problem formulation (e.g., planner requirements, language models used in the domain such as types and objects, plus standard definitions of the domain and problem). A planner then uses this information to check that a solution exists and to generate a plan that satisfies the goal. Using the generated plan, PDSim interprets the action effects as 3D animations and graphics effects to deliver a visual representation of the world and its actions during plan execution and aid the user in assessing the validity of the plan during execution.

While several tools do exist for validating planning models (e.g., plan validation tooks like VAL (Howey and Long 2003) and formal plan verification methods such as (Bensalem, Havelund, and Orlandini 2014; Cimatti, Micheli, and Roveri 2017; Hill, Komendantskaya, and Petrick 2020)), approaches based on visual simulation and visual feedback can also play an important role in addressing the problem of correctly modelling planning domains. Visual tools can serve as powerful environments for displaying, inspecting, and sim-

[1] https://github.com/potassco/pddl-instances/tree/master/ipc-2011/domains/floor-tile-sequential-satisficing

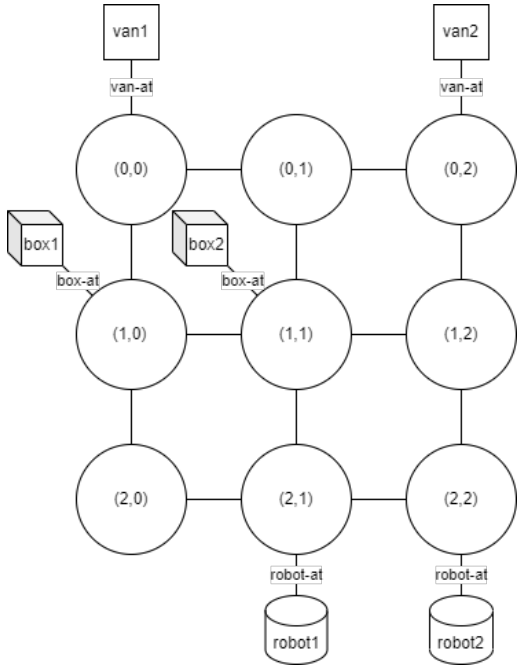

Figure 1: Warehouse planning problem example.

ulating the planning process, which can aid in plan explainability for human users (Fox, Long, and Magazzeni 2017).

In this paper, we describe the aims, structure, and core components of PDSim that are responsible for providing visualisations, and illustrate how PDSim can be used to simulate planning problems. PDSim is built by extending the Unity game engine editor (Unity Technologies 2022) and uses the components offered by the engine such as a path planner, lighting system, and scene management, among others. The system uses a backend server that is responsible for parsing PDDL files and managing plan generation, providing support for a wide range of PDDL language features (such as typing, temporal actions, action cost, etc.). This paper provides a comprehensive description of PDSim, extending earlier versions of the system.

The rest of the paper is organised as follows. First, we review work related to planning problem visualisation and verification. We then describe the structure and main components of PDSim, providing examples of their use in practice by illustrating a number of planning domains. Finally, we conclude with future work and planned additions to PDSim.

Related Work

PDSim (De Pellegrin and Petrick 2022) is part of the small ecosystem of simulators for automated planning which use visual cues and animations to translate the output of a plan into a 3D environment. The closest approach to ours is Planimation (Chen et al. 2020) which uses Unity as the front-end engine to display objects and animate their position while following a given plan. Planimation defines animations using an ad hoc language (namely, an animation profile) similar to PDDL. This differs from PDSim, where animations

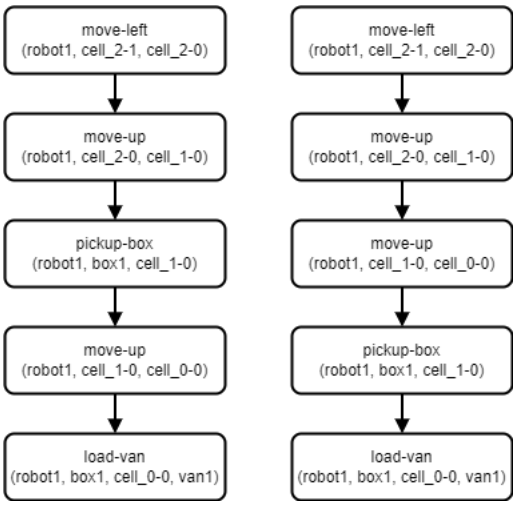

Figure 2: A correct (left) and incorrect (right) plan in the Warehouse domain.

are defined using Unity's visual scripting system.[2]

The Logic Planning Simulator (LPS) (Tapia, San Segundo, and Artieda 2015) also provides a planning simulation system that represents PDDL objects with 3D models in a user-customisable environment. The approach is integrated with a SAT-based planner and a user interface that enables plan execution to be simulated while visualising updates to the world state and individual PDDL properties in the 3D environment. LPS is not based on Unity but provides the user with a simple interface for plan visualisation. Several user-specified files are also required to define 3D object meshes, the relationship between PDDL elements and 3D objects, and the specific animation effects.

vPlanSim (Roberts et al. 2021) is a similar application that also aims to provide a 3D visualization of a plan, but with a number of important differences. While vPlanSim offers a simple and fast custom graphical environment for creating plan simulations with few dependencies, PDSim uses the Unity game engine to offer the user industry-standard tools for creating realistic scenarios. PDSim also provides a language-agnostic tool to set up simulations which is key for users who are not familiar with PDDL and Unity.

Several systems also exist to help users formalise planning domains and problems through user-friendly interfaces. For instance, GIPO (Simpson, Kitchin, and McCluskey 2007), ItSimple (Vaquero et al. 2007) and VIZ (Vodrázka and Chrpa 2010) use graphical illustrations of the domain and problem elements, removing the requirement of PDDL language knowledge, to help new users approach planning domain modelling for the first time. Other tools such as Web Planner (Magnaguagno et al. 2017) and Planning.Domains (Muise 2016) use Gantt charts or tree-like visualisation methods to illustrate generated plans and the state spaces searched by a particular planning algorithm. PlanCurves (Le Bras et al. 2020) uses a novel interface based

[2]https://docs.unity3d.com/Packages/com.unity.visualscripting@1.7/manual/vs-nodes-reference.html

on time curves (Bach et al. 2015) to display timeline-based multiagent temporal plans distorted to illustrate the similarity between states. All of these tools attempt to assist users in understanding how a plan is generated and to help detect potential errors in the modelling process.

Simulators are also prevalent in robotics applications, and multiple systems make use of game engines to provide virtual environments, such as MORSE (Echeverria et al. 2011) or Drone Sim Lab (Ganoni and Mukundan 2017). Game engines also offer several benefits such as multiple rendering cameras, physics engines, realistic post-processing effects, and audio engines, without the need to implement these features from scratch (Ganoni and Mukundan 2017), making them desirable tools for simulation. For example, Unity has been used as a tool for data visualisation, architectural prototypes, robotics simulation (Green et al. 2020), and synthetic data generation for computer vision (James Fort and Davis 2021) and machine learning applications (Haas 2014; Craighead, Burke, and Murphy 2008). There are also interesting use cases of Unity related to AI and planning, including the Unity AI Planner,[3] an integrated planner being created by Unity as a component for developing AI solutions for videogames, and Unity's machine learning agents,[4] a solution for training and displaying agents whose behaviour is driven by an external machine learning component.

Automated Planning Background

Automated planning involves reasoning about a set of actions in order to construct a plan (usually a sequence of actions) that achieves a goal from an initial state (Ghallab, Nau, and Traverso 2004; Haslum et al. 2019). Planning is often thought of as a search through a state space where actions provide a transition system between states.

PDDL (McDermott et al. 1998) provides a standard language for modelling planning problems, by specifying a representation for properties, actions, initial states, and goals (among other features). PDDL splits the planning problem into two parts: the domain, which defines the state properties and the actions; and the problem, which defines the initial state and the goal. A state is a set of all the properties in the planning problem, describing the conditions of the objects or agents at some point in time (Ghallab, Nau, and Traverso 2004). PDDL uses predicates to represent fluents that can be true or false in a state. Predicates are typically parametrized with a set of variable arguments that could be replaced by objects in the problem. For example,

```
at(robot, location)
```

might be used to describe the current location of a robot.

PDDL actions are formalised following a defined schema that specifies the parameters, preconditions, and the effects of each action, as in Figure 3. The preconditions specify the conditions required to perform the action, while the effects describe the changes to the state after an action is performed. Together, the actions capture the state transitions

[3]Unity AI Planner: https://docs.Unity3d.com/Packages/com.Unity.ai.planner@0.0/manual/index.html

[4]Machine Learning Agents: https://github.com/Unity-Technologies/ml-agents

```
(:action
  :parameters (param1, param2, ...)
  :precondition (precondition-formula)
  :effect (effect-formula)
)
```

Figure 3: PDDL action representation.

```
(:action pick-up-box
  :parameters (?r - robot ?b - box ?c - cell)
  :precondition (and
            (robot-at ?r ?c)
       (robot-empty ?r)
            (box-at ?b ?c))
  :effect (and
              (not (robot-empty ?r))
              (not (box-at ?b ?c))
              (robot-has ?r ?b))
)
```

Figure 4: PDDL action for the custom warehouse domain.

```
move(robot_1, office, storage_room)
pick_up(robot_1, box_3, storage_room)
move(robot_1, storage_room, load_bay)
load(robot_1, van_2, box_3)
```

Figure 5: Plan example for Robot warehouse.

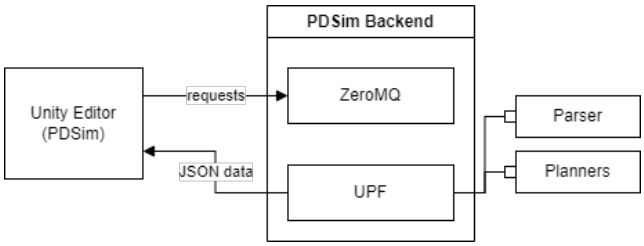

Figure 6: High-level PDSim system architecture.

that are possible in the problem. For example, Figure 4 represents the PDDL action for picking up a box in the custom warehouse domain. The action has three parameters: $?r$, the robot, $?b$, the box, and $?c$, the cell. The precondition specifies the robot must be located at the cell $?c$, the robot must be empty (i.e., not carrying any boxes), and the box $?b$ must be located at the same cell. The effect of the action states that r will no longer be empty, the box will no longer be at the cell $?c$, and the robot will have the box $?b$. PDSim help in visualising these textual representations with animations.

An automated planner can use the PDDL problem representation to generate a plan by choosing actions to sequence together so that the plan achieves the goal from the initial state. For instance, Figure 5 shows a plan for a robot to achieve a goal in a warehouse problem. PDSim uses the PDDL problem representation in order to define and animate several aspects of its visualisation, as described below.

PDSim System Architecture

The high-level structure of the PDSim system is shown in Figure 6. The PDSim system can be imported into Unity3D as a common asset, where the Unity editor interface is used to interact with PDSim components, such as setting the simulation scene, creating animations, or importing 3D or 2D models. PDSim also relies on a Python backend implementation, which is used to parse PDDL files and generate plans. A PDSim simulation is initialised and handled by the backend server running the Unified Planning Framework (UPF; see below), which is responsible for parsing and building a JSON representation of the planning model and running a user-defined planner (defaulting to FastDownward) to generate a plan. UPF is a planner-agnostic framework for Python, which increases PDSim's modularity and lets users select their preferred planner implementation, separating it from the simulation stage itself which comes later in the process. We describe the major components of PDSim below.

PDSim Components

Several PDDL components are key to simulating a planning problem, including: predicates, actions, types (not mandatory), and initial values. A PDDL domain file is used to build the core elements and the animations for the simulation. The types and objects define *SimulationObjects*, the visual aspect of the simulation in Unity: 3D models or 2D sprites. PDDL's predicates are used to define the 2D/3D animations using the visual scripting option that Unity offers. This visual scripting language is used to define common transformation operations, path planning, audio emission, particle effects, etc.

For example, Figure 10 shows an animation definition for the earlier Warehouse planning problem, for a predicate that captures the movement of the robot position from the current grid to an adjacent cell. Action effects are the animated components, where every predicate in the effects list that has an associated animation graph will execute an animation at simulation time. Finally, the problem's initial values are used during simulation time to set up the scene. Similar to the animation effects, all the grounded values are animated if the predicates are associated with an animation.

Backend System

PDSim's backend system is a Python server that communicates with the Unity editor and supports communication between the planning and animation components of the system. In particular, Figure 9 shows the workflow executed by the system when the user wants to create a new simulation. The user interacts with PDSim using the Unity editor to specify the planning domain and problem files. Unity tries to connect to the backend server by submitting a request using these files. The planner interface can use a local planner, such as FastDownward (Helmert 2006) (the default planner), or the planning web service offered by Planning.Domains (Muise 2016). If either the parsing or planning actions fail, the interface will warn the user of the error.

PDDL domain and problem elements are converted to a JSON representation and sent back to Unity, which will create objects and animations that can be customised by the user. Domain entities are used to set up the core Unity simulation. For instance, Figure 7 shows the JSON code used to establish the internal definitions of actions, types, and predicates for the logistics domain. Problem entities are used to set up a Unity-level scene, as in Figure 8. Once these components are defined, the user can customise them using the Unity editor, for instance configuring multiple problems for the same domain, or multiple simulations for different plans.

```
{ 'predicates':
  {'in-city':
      {'args': ['place', 'city'],
       'arity':2}, ... },
 'actions':
  {'load-truck':
     {'effects':[
        {'args'['pkg', 'loc'],
         'fluent': 'at',
         'negated': true} ... ],
      'params':{'pkg':'package',
                'truck':'truck',
                'loc':'place'} } ... }
 'types':
  {'object':['city','place','physobj'],
   'place':['airport','location'] ...}
}
```

Figure 7: Example domain representation in JSON.

```
{'objects':
  {'apn1' : 'airplane'
   'city1' : 'city', ... },
 'init':
  {'at': [
     {'args': ['obj13', 'pos1']
      'value': true},
     {'args': ['obj23', 'pos2']
      'value': true}, ... ] ... }
}
```

Figure 8: Example problem representation in JSON.

At the technical level, communication between PDSim's backend server and Unity is provided by the ZeroMQ networking library,[5] in particular the Python implementation package pyzmq[6] on the server side and the C# implementation netMQ[7] on the Unity side.

Unified Planning Framework (UPF)

PDSim's backend system wraps the functionality of the Unified Planning Framework (UPF) as the main tool for manipulating and solving planning problems in PDSim. UPF is a Python library provided by the AIPlan4EU project[8] that aims to simplify the use of automated planning tools for AI application development. UPF attempts to standardize aspects of the planning process, making it accessible to

[5] https://zeromq.org/

[6] https://pypi.org/project/pyzmq/

[7] https://github.com/zeromq/netmq/

[8] https://www.aiplan4eu-project.eu/

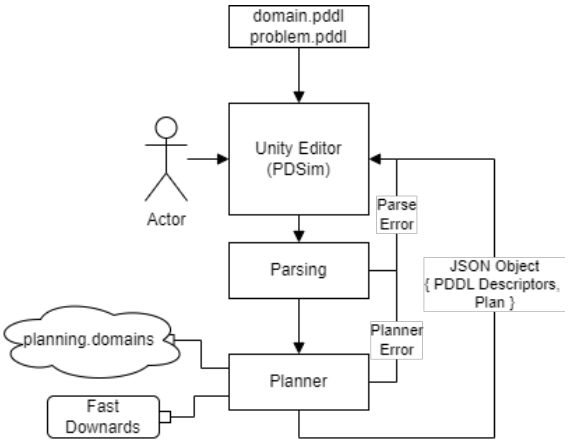

Figure 9: Simulation workflow in PDSim.

users of any level of expertise. In particular, it offers a well-developed PDDL parser and a standard interface for communicating with external planners. Integration with UPF enables the PDSim system to take advantage of these features and any future updates that UPF may provide.

Planning Domain Simulation in Unity

Unity (Unity Technologies 2022) is a popular state-of-the-art game engine used for building 3D projects across a range of diverse applications. In PDSim, Unity provides the front-end interface and is responsible for handling all of the 2D/3D graphics and animations related to the simulation.

One of the fundamental design concepts used by Unity is the idea of *composition*, which means that an object can be *composed* of different types of objects. In particular, Unity's component system provides the capability for every object in a Unity scene to be assigned custom scripts or modules, such as a rigid body for the physics simulation, a collision volume, an audio source, etc. Every object in Unity can also be scripted using the C# language, meaning that an object can have a user-defined behaviour in the scene. For example, an object can respond to user inputs from a mouse or keyboard, or can be translated, rotated and scaled, or have its colour changed, based on conditional events. Object scripting in Unity is key to the modularity of the simulation, especially for the custom representation of PDDL elements.

Scripting can also be applied to the editor window, where users interact with the engine and where it is possible to set the properties of the objects in the scene by using Unity's user interface. PDSim make heavy use of all the features provided by Unity, such as the Visual Scripting Language used to create animations and events. As a result, users do not need to learn a new language for developing animations, and animation graphs can be modified on-the-fly without waiting for scripts to be recompiled.

PDDL to PDSim

To use PDDL in the Unity Game Engine, there is a need to translate the PDDL specification into a format that can

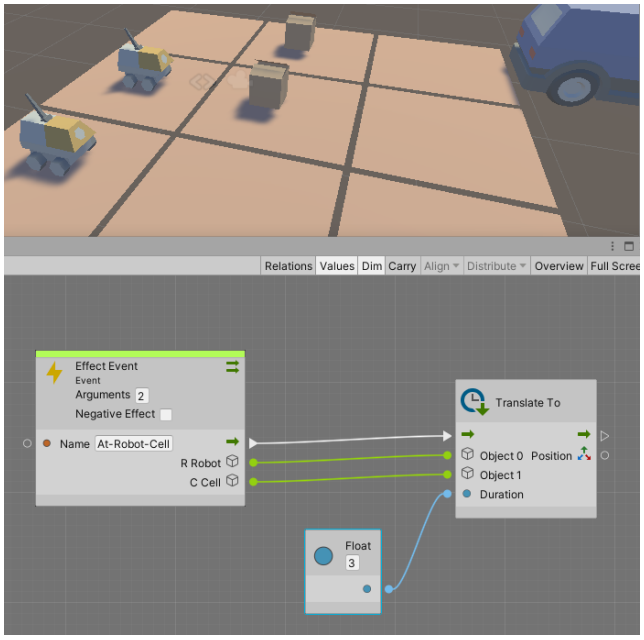

Figure 10: Animation definition example.

be used within the Unity environment. This involves creating custom C# classes and objects that represent the knowledge specified in the PDDL domain. Taking the warehouse domain example, the PDDL specification includes a set of objects, such as robots, boxes and vans. There is a need to create corresponding C# classes to represent these objects. Including properties to represent the various attributes of the objects, such as their transform (position, rotation and scale), colour, and other relevant visualization features. In addition, methods to perform various operations on the objects and perform updates on the object attributes. Similarly, PDDL predicates and actions need to be mapped using C# dictionaries to animations that visualise a change in the world state.

Simulation Objects

A PDDL type in PDSim is represented by a *simulation object*, a structure that shares similar information for all the objects defined in a planning problem. A simulation object is defined by two main components: models and control points. Models are used to visually represent the object type in the virtual world (e.g., block, airport, player, robot, etc.). These can be 3D meshes or 2D textured sprites that can be imported in the Unity editor. A user can add as many models as they like. A collision box that wraps all the models is automatically calculated to be used later in the simulation to detect the interaction with the user inputs and the collisions calculated by the physics engine. Control points are 3D vectors that represent particular points of interest in the object type representation (e.g., the cardinal points of an object, a point that represents the arm position of an agent, etc.).

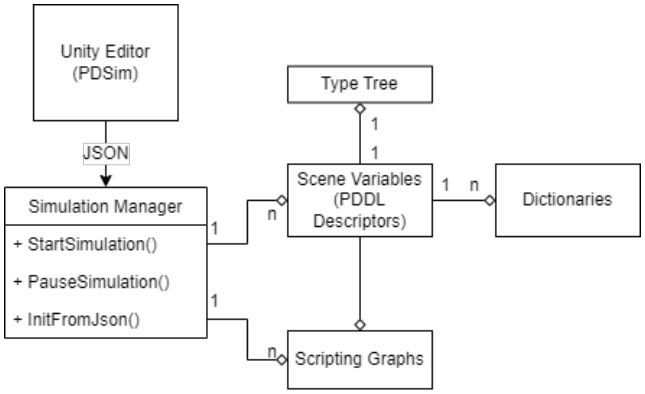

Figure 11: Simulation Manager diagram.

```
{'status' : 'OK',
'plan':[
{
    'action': 'pick-up',
    'attributes' : ['b']
},
{
    'action': 'stack',
    'attributes' : ['b', 'a']
},
{
    'action': 'pick-up',
    'attributes' : ['c']
}, ...]}
```

Figure 12: A parsed plan in JSON format.

Animations

One of the most important aspects of PDSim is the visual script animation system. As shown in Figure 10, users can create their own particular behaviour in the virtual scene for every predicate they want to animate. The example shows a simple translation animation from an object position to a target position. In particular, the example shows one of the custom animation nodes developed in PDSim to help simplify the creation of animations for new users. Every predicate in an action's effect can have one of these graphs linked to it, and every graph comes with an *EffectEvent* that is invoked during plan simulation with the corresponding objects from the Unity scene (i.e., the objects in the plan's action).

To simplify the development of new animations, and to help new users with visual scripting, a set of predefined animation nodes has been created which cover a number of useful simulation cases that frequently arise, such as:

1. **TranslateTo**: An animation for moving a particular object in the scene to a specific point in the world or to another object position.
2. **RotateTo**: An animation to rotate a particular object in the scene to an angle or to look at another object in the world.
3. **PathTo**: An animation for moving an object using Unity's path planning system.
4. **Spawn**: A node to instantiate an object (i.e., a 3D mesh) in the scene.
5. **GetObjectFromScene**: A node mainly used to access the components added by the user that aren't part of the PDDL definition.

These predefined animation nodes aim to reduce the size and complexity of the scripting graph, and simplify the animation process for new Unity users. From a technical point of view, these animations use Unity *Coroutines* that allow the user to write functions that can run concurrently in the main Unity thread and be suspended or resumed either by user choice or if a condition is met.

Simulation Manager

Figure 11 shows a diagram of the simulation manager, a component that handles simulations on the front-end side. The main responsibilities of the simulation manager include:

- Starting or pausing the simulation (during plan time),
- Holding references of all the PDDL descriptors in the scene (predicates, actions, objects),
- Holding references of all animation graphs defined by the user,
- Keeping track of the existing types (if defined) in the domain file, and
- Sending request to the backend server to update or initialize the PDDL representation.

If types are specified in the PDDL domain file, then the simulation manager creates simulation object blueprints for all the leaf types of the type tree that is built when the domain is parsed for the first time. These types are replicated for each object in the PDDL problem file that matches the particular type, using the user configuration of simulation objects, as described above.

The simulation manager is initialised using the JSON data from the backend server containing the PDDL elements and the representation of the plan, as shown in Figure 12, using a plan for a Blocks World domain problem. Every action effect will have an associated list of animation graphs representing the effect of the PDDL action. The simulation manager will execute the animations using the attributes in the plan representing the simulation objects involved in the simulation of that action. As the first step in every simulation, PDDL's *init* block is animated. Init represents the starting state of a planning problem and is defined by a list of fluents describing the current state of the world. These fluents are represented in the form of *fluent_name(arguments)* where the arguments are the objects that are present in the environment. The simulation manager will publish events with the corresponding fluent name and objects from the simulation scene that will be used by the visual scripting language to map which animation to execute and the graphical objects to use. The same process is then repeated for every action effect from the plan.

Planning Domain Examples

PDSim has been tested using the published benchmark domains from the International Planning Competition (IPC).[9] We illustrate the capabilities of PDSim to simulate planning problems using the Blocks World, Logistics, and Sokoban domains, plus the custom Warehouse domain introduced earlier in the paper.

Blocks World: The Blocks World domain (IPC 2000) is one of the simplest planning domains: blocks can be stacked on top of each other and only one block can be picked, moved, and dropped at a time. The goal is achieved when the specified stack sequence is reproduced.

Figure 13 shows an example of a Blocks World action sequence being simulated in PDSim. The three snapshots represent different steps in a plan and demonstrate the interaction with objects in the scene. The user interface shows the transition of fluents that describe an object after the action effects are applied during plan execution. In the example, the object *d* starts with the fluent *on table* describing the initial condition of the problem for that object. The third image in Figure 13 shows how the fluents change after the plan has finished executing.

Logistics: The Logistics domain (IPC 2000) describes a problem involving packages that need to be transported between cities using an aeroplane, and within cities using trucks. This domain steps up the complexity of the simulation environment while keeping simple definitions of predicates and actions (e.g., the predicates *InCity*, *At*, *In* are used to respectively describe if a location is inside a city, if an object is in a location, and if a package is in a vehicle). Figure 15 shows a snapshot of the Logistics simulation, highlighting the animation of boxes and aeroplanes.

Sokoban: The Sokoban domain (IPC 2008) describes the Sokoban game problem,[10] where a player needs to move an object to a predefined goal on a grid. Figure 16 illustrates a typical problem level for Sokoban with a player and stone that need to be moved. This domain adds to the complexity of the previous example, illustrating the functionality of this simulation in Unity and its ability to rapidly provide an in-game agent.

Custom Warehouse Domain: To demonstrate the use of user-defined domain models, the warehouse planning problem presented in the introduction has been simulated in PDSim as illustrated in Figure 14. Both correct and incorrect domain models have been tested with PDSim, showing the robot picking up the box in the correct cell position and executing the non-intuitive action of picking up the box from the wrong cell (as represented in the images).

Real World Robot Application: PDSim has been used as part of a robotics project involving robots acting in a human environment. Figure 17 show the simulation for a custom planning problem involving a robot operating in a flat equipped with a different type of sensors. PDSim can simulate the change in the environment state (as shown in the bottom image).

Figure 17: PDSim using Human Support Robot (Yamamoto et al. 2019).

Discussion

In general, PDSim offers a powerful and flexible framework for visualising planning problems using a state-of-the-art graphical engine. More specifically, PDSim aims to fill a gap in current systems that provide plan simulations, by offering users a simplified environment to develop 3D or 2D simulations, compared with current approaches that come with the overhead of learning and using an ad hoc scripting language to interact with a custom simulator (Tapia, San Segundo, and Artieda 2015; Chen et al. 2020; Roberts et al. 2021). PDSim is designed as a support system for automated planning by providing intuitive tools to interface with a plan solution. One main limitation of PDSim is the lack of an automated and formal validation tool for planning, which could be integrated in the future. However, our approach offers a practical and human-centred way to check if a plan is valid and can be executed (Howey and Long 2003) using a graphical engine to gamify the process of validating a planning solution. Approaches like (Le Bras et al. 2020; Fox, Long, and Magazzeni 2017) also suggest that answering the question of why an action has been successfully executed or has failed, further increases the explainability of a plan. In this context, PDSim provides intuitive hints about possible errors using visual cues, by displaying an interface with the transitions of each action and how they modify the state of a particular object or agent. It is important to reiterate, however, that PDSim is primarily aimed at planning-agnostic users like students. Within this group, as (Chen et al. 2020) indicates, there is a difference between the mental model the user has of the planning problem and the actual implementation. In fact, PDDL is often approached as a traditional program-

[9] https://github.com/potassco/pddl-instances

[10] https://en.wikipedia.org/wiki/Sokoban

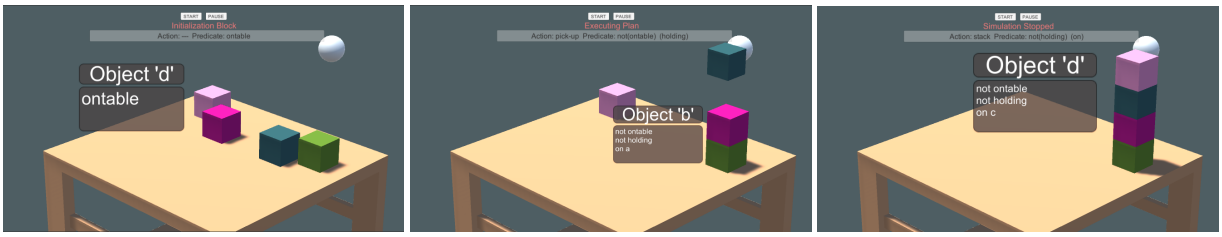

Figure 13: PDSim Blocks World simulation sequence.

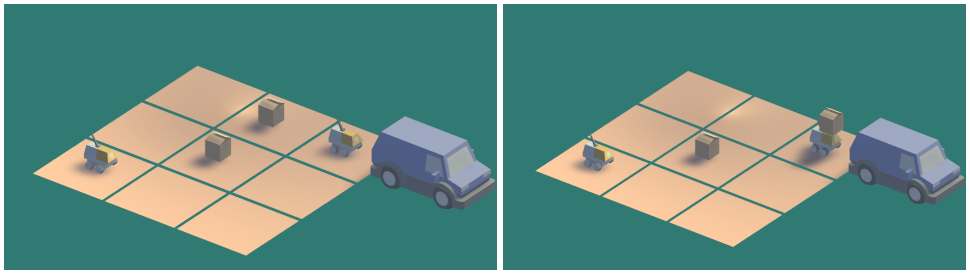

Figure 14: PDSim Warehouse Robot simulation with correct (left) and incorrect (right) action simulation.

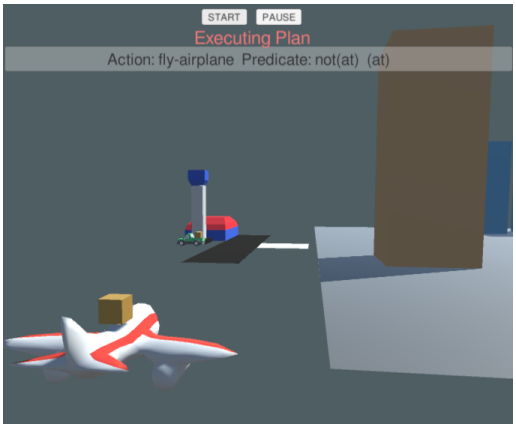

Figure 15: PDSim Logistics simulation.

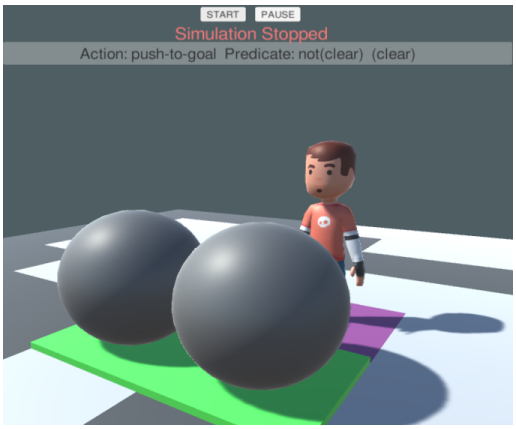

Figure 16: PDSim Sokoban simulation.

ming language by beginners, rather than a knowledge definition language. With this in mind, PDSim aims to simplify the learning curve of PDDL by assisting with components that provide information about the state of planning entities in real-time.

Conclusion and Future Work

This paper presented the structure and operation of PDSim, a simulation system for animating PDDL-based planning domains and plans. PDSim uses the Unity game engine to animate PDDL predicates through an action's effects, by linking a visual scripting graph to each of them. The user can modify and customise the behaviour by effectively programming an animation from scratch or by quickly using the high-level animations provided. Created simulations can be exported in an executable and, given the cross-platform nature of Unity, all major operating systems can be targeted (e.g., Windows, Mac, Linux).

As future work, we plan to introduce a more intuitive way to create and modify the knowledge model, using the same visual scripting paradigm, and thus completely removing the need to know the PDDL language syntax. This will be internally used together with an in-engine planner that the user can interact with at planning time to change object properties and replan on the fly. Given the close relationship between PDSim and Unity, it will also be possible to use applications such as extended reality (XR) to interact with the plan. Another planned direction for PDSim will also be to include extensions for visualising the current state of an agent's knowledge and beliefs to support epistemic planning, allowing visualisations to be generated from different agent perspectives. Finally, we also plan to evaluate the use of PDSim in an education setting, and gather feedback about the overall helpfulness and usefulness of PDSim as a development aid for students learning about automated planning in an intro-

ductory AI course. To further simplify the accessibility of PDSim, we think that an evaluation with a group of students is needed to gather feedback about the overall helpfulness and easiness of use in respect of the standard approach to planning with PDDL.

References

Bach, B.; Shi, C.; Heulot, N.; Madhyastha, T.; Grabowski, T.; and Dragicevic, P. 2015. Time curves: Folding time to visualize patterns of temporal evolution in data. *IEEE transactions on visualization and computer graphics*, 22(1): 559–568.

Bensalem, S.; Havelund, K.; and Orlandini, A. 2014. Verification and validation meet planning and scheduling. *International Journal on Software Tools for Technology Transfer*, 16: 1–12.

Chen, G.; Ding, Y.; Edwards, H.; Chau, C. H.; Hou, S.; Johnson, G.; Sharukh Syed, M.; Tang, H.; Wu, Y.; Yan, Y.; Gil, T.; and Nir, L. 2020. Planimation.

Cimatti, A.; Micheli, A.; and Roveri, M. 2017. Validating domains and plans for temporal planning via encoding into infinite-state linear temporal logic. In *Proceedings of AAAI*, 3547–3554.

Craighead, J.; Burke, J.; and Murphy, R. 2008. Using the unity game engine to develop sarge: a case study. In *Proceedings of the 2008 Simulation Workshop at the International Conference on Intelligent Robots and Systems (IROS 2008)*.

De Pellegrin, E. 2020. PDSim: Planning Domain Simulation with the Unity Game Engine.

De Pellegrin, E.; and Petrick, R. P. 2021. Automated Planning and Robotics Simulation with PDSim.

De Pellegrin, E.; and Petrick, R. P. 2022. What Plan? Virtual Plan Visualization with PDSim.

Echeverria, G.; Lassabe, N.; Degroote, A.; and Lemaignan, S. 2011. Modular open robots simulation engine: Morse. In *2011 IEEE International Conference on Robotics and Automation*, 46–51. IEEE.

Fox, M.; Long, D.; and Magazzeni, D. 2017. Explainable Planning. In *Proceedings of the IJCAI Workshop on Explainable AI*.

Ganoni, O.; and Mukundan, R. 2017. A framework for visually realistic multi-robot simulation in natural environment. *arXiv preprint arXiv:1708.01938*.

Ghallab, M.; Nau, D.; and Traverso, P. 2004. *Automated Planning: theory and practice*. Elsevier.

Green, C.; Platin, J.; Pinol, M.; Trang, A.; and Vij, V. 2020. Robotics simulation in Unity is as easy as 1, 2, 3!

Haas, J. K. 2014. A history of the Unity game engine.

Haslum, P.; Lipovetzky, N.; Magazzeni, D.; and Muise, C. 2019. An introduction to the planning domain definition language. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 13(2): 1–187.

Helmert, M. 2006. The Fast Downward Planning System. *Journal of Artificial Intelligence Research*, 26: 191–246.

Hill, A.; Komendantskaya, E.; and Petrick, R. P. A. 2020. Proof-Carrying Plans: A Resource Logic for AI Planning. In *International Symposium on Principles and Practice of Declarative Programming (PPDP)*, 1–13.

Howey, R.; and Long, D. 2003. VAL's Progress: The Automatic Validation Tool for PDDL2.1 used in the International Planning Competition. In *Proceedings of the ICAPS Workshop on The Competition: Impact, Organization, Evaluation, Benchmarks*.

James Fort, J. H.; and Davis, N. 2021. Boosting computer vision performance with synthetic data.

Le Bras, P.; Carreno, Y.; Lindsay, A.; Petrick, R. P. A.; and Chantler, M. J. 2020. PlanCurves: An Interface for End-Users to Visualise Multi-Agent Temporal Plans. In *Proceedings of the ICAPS Workshop on Knowledge Engineering for Planning and Scheduling (KEPS)*.

Magnaguagno, M. C.; Fraga Pereira, R.; Móre, M. D.; and Meneguzzi, F. R. 2017. Web planner: A tool to develop classical planning domains and visualize heuristic state-space search. In *ICAPS Workshop on User Interfaces and Scheduling and Planning (UISP)*.

McDermott, D.; Ghallab, M.; Howe, A.; Knoblock, C.; Ram, A.; Veloso, M.; Weld, D.; and Wilkins, D. 1998. PDDL—The planning domain definition language. Technical Report CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control.

Muise, C. 2016. Planning.domains. ICAPS System Demonstration.

Roberts, J. O.; Mastorakis, G.; Lazaruk, B.; Franco, S.; Stokes, A. A.; and Bernardini, S. 2021. vPlanSim: An Open Source Graphical Interface for the Visualisation and Simulation of AI Systems. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 31, 486–490.

Simpson, R. M.; Kitchin, D. E.; and McCluskey, T. L. 2007. Planning domain definition using GIPO. *The Knowledge Engineering Review*, 22(2): 117–134.

Tapia, C.; San Segundo, P.; and Artieda, J. 2015. A PDDL-based simulation system. In *Proceedings of the IADIS International Conference Intelligent Systems and Agents*.

Unity Technologies. 2022. Unity.

Vaquero, T. S.; Romero, V.; Tonidandel, F.; and Silva, J. R. 2007. itSIMPLE2.0: An Integrated Tool for Designing Planning Domains. In *Proceedings of ICAPS*, 336–343.

Vodrázka, J.; and Chrpa, L. 2010. Visual design of planning domains. In *Proceedings of ICAPS Workshop on Knowledge Engineering for Planning and Scheduling (KEPS)*, 68–69.

Yamamoto, T.; Terada, K.; Ochiai, A.; Saito, F.; Asahara, Y.; and Murase, K. 2019. Development of human support robot as the research platform of a domestic mobile manipulator. *ROBOMECH journal*, 6(1): 1–15.