

Plan Simulation with PDSim

Emanuele De Pellegrin¹, Ronald P. A. Petrick¹

¹Edinburgh Centre for Robotics, Heriot-Watt University, Edinburgh, Scotland, United Kingdom

Abstract

This paper reports on 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. The project aims to fill a gap in the area of planning simulation and validation by tackling the problem of the scarcity of systems and tools to help the user quickly evaluate the validity of the planning model. Simulating a planning problem using 3D graphics and animation techniques can help the user to quickly evaluate the quality of a plan and improve the design of the planning domain and problem. This paper presents an overview of PDSim, including its aims as a system for automated planning, the current state of development, and future plans for the project.

Keywords

Automated Planning, Unity Game Engine, Simulation

1. Introduction

Verifying plan solutions and debugging planning domain models can be quite challenging, especially for real-world planning problems that involve a large number of actions and objects. While languages like PDDL [1] provide a standard way of representing planning models supported by a range of planners, catching modelling errors (i.e., incorrect logic in action effects and preconditions, 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. Although several tools do exist to aid in the validation of planning domain models (e.g., VAL [2]), and formal plan verification methods are a growing area of research [3, 4, 5], approaches based on visualisation methods and visual feedback can also play an important role in addressing the problem of correctly modelling planning domains. Visual tools can also serve as an environment for displaying, inspecting, and simulating the planning process, which can aid in plan explainability for human users [6].

PDSim (Planning Domain Simulation) [7] introduced a system to visualise and simulate plans for classical planning problems defined in PDDL. While visualisation of planning domains and plan solutions is not a new idea [8, 9, 10, 11, 12], PDSim approaches the problem by building a graphical environment for plan visualisation and simulation within the Unity game engine [13]. PDDL is used to define the structure of the domain knowledge and the problem formulation (e.g., planner requirements, language models used in the domain such as types and objects, plus

IPS 2021: 9TH ITALIAN WORKSHOP ON PLANNING AND SCHEDULING, November 29–30, 2021


✉ ed50@hw.ac.uk (E. D. Pellegrin); R.Petrick@hw.ac.uk (R. P. A. Petrick)

🌐 <https://cryoscopic-e.github.io/> (E. D. Pellegrin); <http://petrick.uk/> (R. P. A. Petrick)

🆔 0000-0002-5979-6547 (E. D. Pellegrin); 0000-0002-3386-9568 (R. P. A. Petrick)



© 2021 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

 CEUR Workshop Proceedings (CEUR-WS.org)

standard definitions of the domain and problem) in a 3D visual setting. These components are used by a planner to check that a solution exists and to generate a list of actions (a plan) satisfying the goal. Using the plan, PDSim interprets the action effects as 3D animations and graphics effects that can be used to assess the validity of the plan and to deliver a visual explanation of the world and its actions during plan execution. The main “actors” involved in PDSim simulations are therefore the properties responsible for defining the initial state and the action effects.

In this paper, we describe the aims of PDSim, the structure and core components that are responsible for providing virtual simulations, and illustrate how PDSim can be used to simulate classical planning problems. PDSim is built by extending the Unity game engine editor [13] and uses the components offered by the engine such as a path planner, lighting system, and scene management, among others. The system uses a back-end server that is responsible for parsing PDDL files and managing plan generation. The parser supports a wide range of PDDL language features. Thanks to the modular nature of the server side, the system is also extensible: one planned extension is support for the Robot Operating System (ROS) [14], enabling the use of PDSim as a visualisation tool for robotics and related applications.

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

2. Related Work

PDSim [7] 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 [10] which uses the Unity game engine as the front end 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 are defined using a custom visual scripting system.

The Logic Planning Simulator (LPS) [11] also provides a planning simulation system that represents PDDL objects with 3D models in a user-customizable 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.

Several systems also exist to help users formalize planning domains and problems through user-friendly interfaces. For instance, systems like GIPO [15], ItSimple [9] and VIZ [16] 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 software such as Web Planner [17] and Planning.Domains [18] use Gantt charts or tree-like visualisation methods to illustrate the generated plan and the state space searched by a particular planning algorithm. PlanCurves [12] uses a novel interface based on time curves [19] to display timeline-based multiagent temporal plans distorted to illustrate similarity between states. All of these tools attempt to assist users in understanding how a plan is generated and to

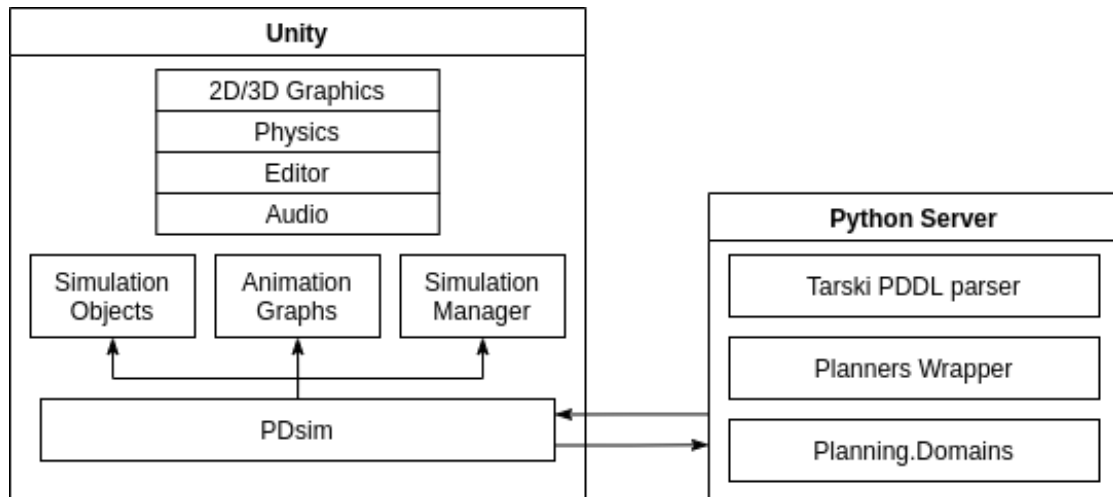


Figure 1: PDSim system design

help detect potential errors in the modelling process.

Simulators are also prevalent in robotics applications, and multiple systems exist that make use of game engines to provide a virtual environment, such as MORSE [20] or Drone Sim Lab [21]. Game engines also offer several benefits such as multiple rendering cameras, physics engines, realistic post-processing effects, and audio engines, with no need to implement these features from scratch [21], making them desirable tools for simulation. For instance, Unity is being used as a tool for data visualization, architectural prototypes, robotics simulation¹, computer vision² and machine learning applications [22, 23]. Interesting use cases of Unity related to AI and planning include the Unity AI Planner³, an integrated planner being created by Unity as a component for developing AI solutions for videogames, and Unity’s machine learning agents⁴, a solution for training and displaying agents whose behaviours are driven by an external python machine learning component.

3. System Implementation

We begin by presenting an overview of PDSim, its structure, and how it is integrated with Unity. Figure 1 presents the overall system design of PDSim. In the following sections we consider each component in more detail and how it is implemented in the system. In general, PDSim can be imported into the Unity engine as a common asset. The simulation is initialized and handled by a python back-end server which is responsible for parsing and building a representation of the planning model, making it easier for Unity to understand.

Table 1 (left) shows the PDDL components used to simulate a planning problem and (right)

¹Unity Robotics: <https://unity.com/solutions/automotive-transportation-manufacturing/robotics>

²Unity for CV: <https://unity.com/products/computer-vision>

³Unity AI Planner: <https://docs.unity3d.com/Packages/com.Unity.ai.planner@0.0/manual/index.html>

⁴Machine Learning Agents: <https://github.com/Unity-Technologies/ml-agents>

Table 1
PDDL to Unity conversion table

PDDL Element	Unity Representation
predicate	Animation graphs define object behaviour in Unity (translation, path planning, audio emission, particle effects, etc.)
action	Action effects are the animated components, using the object attributes
objects	General 2D/3D model representation with custom sprites/meshes
init	Pre-simulation using the predicate's defined animations

```
{'objects':
  [{'name': 'apn1', 'type': 'airplane'},
   {'name': 'apt1', 'type': 'airport'},
   ...],
 'predicates':
  [{'name': 'in-city', 'attributes': ['place', 'city']},
   {'name': 'at', 'attributes': ['physobj', 'place']},
   {'name': 'in', 'attributes': ['package', 'vehicle']}],
 'init':
  [{'predicate': 'at', 'attributes': ['obj13', 'pos1']},
   {'predicate': 'at', 'attributes': ['obj23', 'pos2']},
   ..]
 ..
}
```

Figure 2: Parsed PDDL to JSON example

how Unity uses those components to represent PDDL in a graphical environment. PDDL files are translated into a JSON map of the components needed for simulation. For example, Figure 2 shows the JSON code for the logistics domain. In Unity, the user can set the 2D or 3D models for the constants defined in the planning problem, create animations for specific predicates and use Unity's internal components such as the physics engine, the planning system, etc.

3.1. Back-end

The back-end of PDSim is a python server that communicates with the Unity editor using the ZeroMQ networking library⁵, in particular the python implementation package pyzmq⁶ on the server side and the C# implementation netMQ⁷ on the Unity side. The Figure 3 shows the workflow executed by the system when the user wants to create a new simulation. The user interact with PDSim that use the Unity editor to create a form where the domain and problem

⁵<https://zeromq.org/>

⁶<https://pypi.org/project/pyzmq/>

⁷<https://github.com/zeromq/netmq/>

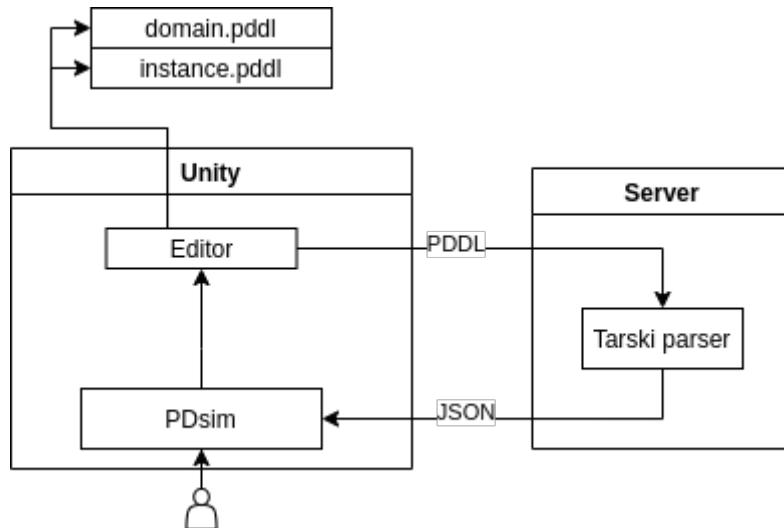


Figure 3: Creating new simulation diagram.

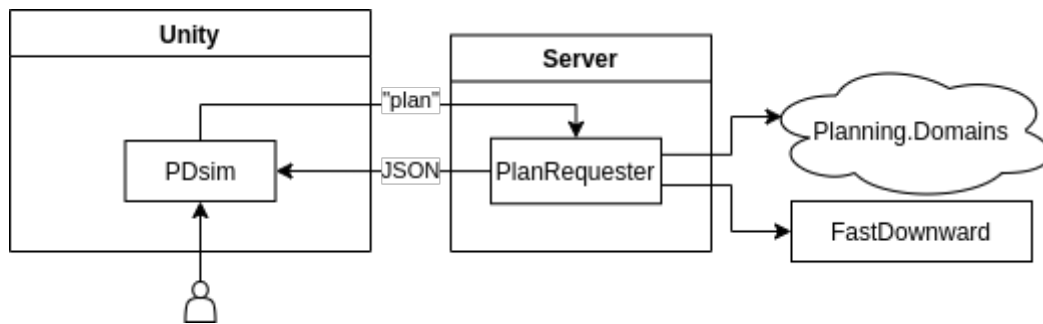


Figure 4: Requesting a new plan diagram.

files are expected. Unity tries to connect, using a separate thread, with the back end server by submitting a request using the PDDL domain and problem files. The request is sent with the "init" header to tell the server that the request is to parse the PDDL and to create a representation on the server of the planning problem to be used for later requests. The server handle the PDDL parsing using the well know Tarski parser ⁸. The PDDL elements are converted to a JSON representation and send back to Unity that will create the objects and animations customizable by the user. As shown in Figure 4, the python server can accept the "plan" request used to generate a plan with the current domain and problem previously initialised. The PlanRequester interface can use a local planner such as FastDownward [24] if present, or the planner web service offered by Planning.Domains [18]. The response sent to Unity use the JSON format as before, representing the actions to animate in PDSim.

⁸<https://github.com/aig-upf/tarski>

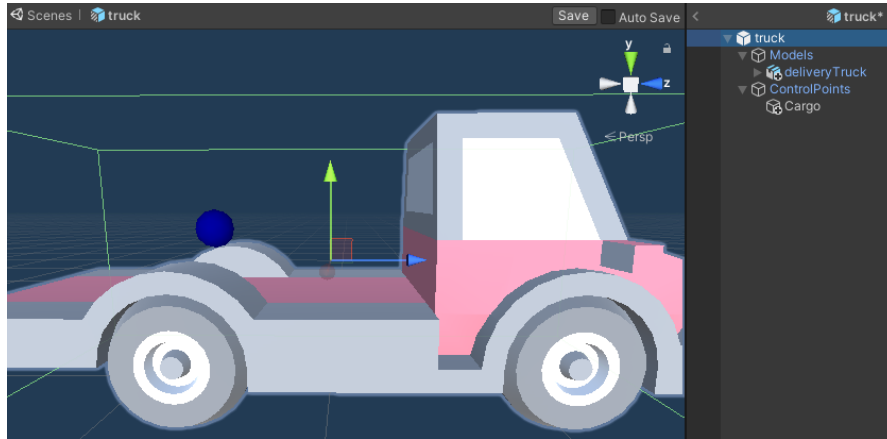


Figure 5: Simulation object example with a truck type from the logistic domain.

3.2. Unity

Unity [13] is a well-known state-of-the-art game engine. Unity is used as the front end of PDSim, and it's responsible for handling all the 2D/3D graphics and animations related to the simulation. One of the fundamental OOP design patterns used by Unity is the *composition*, which means that an object can be *composed* of different types of objects. Thanks to the components system, every object in a Unity scene can have 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 be scripted using the C# language, meaning that an object can have a user-defined behavior in the scene. For example, an object can respond to user inputs such as mouse and keyboards, translate, rotate, and scale, change color, based on conditional events. Being able to script every object in Unity is of utmost importance for the modularity aspect of a simulator, in particular for the custom representation of the PDDL elements. Scripting can also be applied to the editor window itself. The editor window is where the user interacts with the engine, where it's possible to set game objects' properties in the scene by using the Unity's UI.

3.2.1. Simulation Objects

A PDDL type in PDSim is represented by a `SimulationObject`, a prototype that shares the same structure for all the objects defined in the problem. A simulation object is defined by two main components: `Models` and `Control points`. `Models` are used to visually represent in the virtual world the object type (i.e. block, airport, player, robot, etc). These can be 3D meshes or 2D textured sprites that can be imported in the Unity editor. It's possible to add as many models as the user wants, 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 (i.e. the cardinal points of an object, a point that represents the position of the arm of an agent, etc.). Figure 5 shows an example of how a simulation object can be composed. The `Models` is composed only by one mesh representing a delivery truck, and

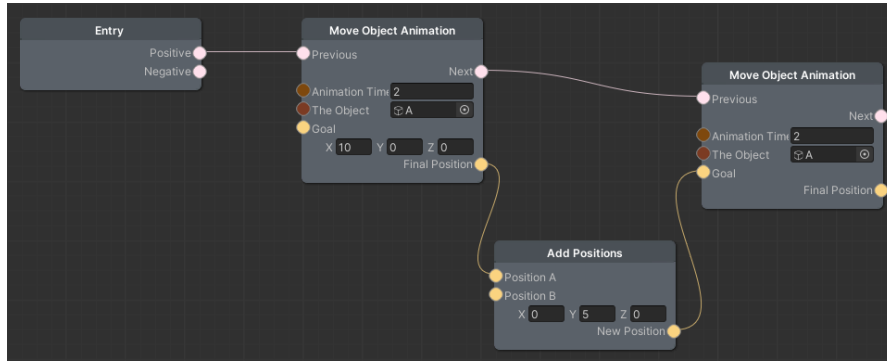


Figure 6: Animation definition example.

Table 2

PDSim animation nodes

Animation	Behaviour
MoveObject	Animation for moving a particular Object in the scene to a specific Point in the world.
ObjectPathTo	Animation for moving an object using Unity's path planning system (Requires setup).
Spawn	Instantiate an Object in the scene.
PlayAudio	Play an audio clip.
Utility Node	Behaviour
AddPositions	The output is a vector sum of the 2 input Vectors If connected to a node, the inputs can be current objects positions (Used to express custom paths)
GetSimObject	Node mainly used to get the components of a simulation object previously defined. Used during run-time to get the current object components involved in an action.

the control point is the 3D vector position of the cargo represented by the blue dot in the scene. The collision box calculated is represented by the green line that contains all the models.

3.2.2. Animations

One of the most important aspects of PDSim is the visual scripting animation system. As shown in Figure 6, the user can create his own particular behaviour in the virtual scene for every predicate he wants to animate. The example shows a stacking of 2 translation animations where the same object is translated along the x axis first and the y axis subsequently. Every animation has a duration, where the value can range from 0 to n seconds and can reference the previous and the next animation. Table 2 shows the animation nodes currently implemented in PDSim.

In Unity these animations use *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.

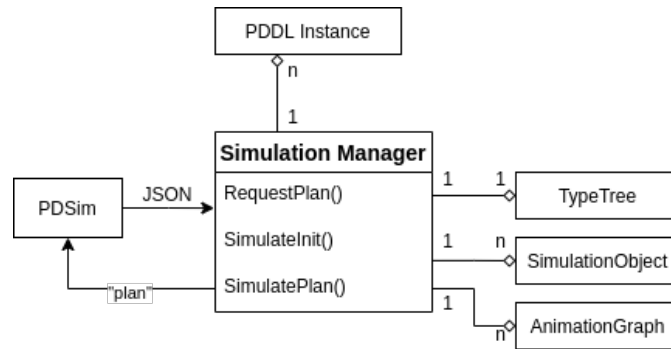


Figure 7: Simulation Manager diagram.

```

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

Figure 8: Parsed plan in JSON format.

3.2.3. Simulation Manager

Figure 7 show the overall diagram of the simulation manager, a component that handles the simulations on the front-end side. Its main responsibilities includes:

- Holding references to the several instances to simulate
- Holding reference of all simulation objects in the scene
- Holding reference of all animation graph defined by the user
- Keep track of the existing types defined in the domain file
- Send request to the back-end server to generate a new plan if the instance changes and no plan exist.

The simulation manager build simulation objects blueprints for all the leaf type of the type tree that is build when the domain is parsed for the first time. These types prefabs are replicated for


```
(pick-up b)
(stack b a)
(pick-up c)
(stack c b)
(pick-up d)
(stack d c)
```

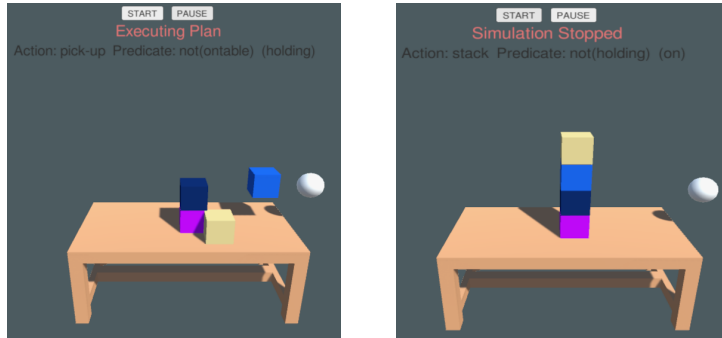


Figure 9: Problem representation for Blocks World

each object in the problem files that match the particular type, using the user configuration explained in Section 3.2.1. The simulation manager communicates with the back-end server to request a plan using the current instance being simulated and the domain corpus active on the server side. The request is a simple message containing the problem file with a *plan* header. The server will respond with a JSON representation of the plan as shows in Figure 8, using the plan for the blocks word domain problem. Every action will have an associated list of animation graphs representing the effect of a PDDL action. The simulation manager will stack and execute those animations using the attributes in the plan representing the simulation objects involved in the simulation of that action.

4. Tested Domains

The main functionality of the system has been tested using particular PDDL benchmark domains, released for the IPC and available in a public repository⁹. The domains selected are Blocks World, Logistics and Sokoban. These particular domains were selected for testing the system by increasing the complexity of the domain, starting with blocks world as the simplest to Sokoban the hardest. To test Unity's internal components and how they integrate into PDSim, a custom domain called Robots and Boxes have been used also to demonstrate how the simulator can be used with user-defined domains.

4.1. Blocks World

Blocks World (IPC 2000) is one of the most famous domains. In this domain, blocks can be stacked one on top of the other. The agent hand can only pick, move and drop one block at a time. The goal is achieved when the specified stack sequence is reproduced. Figure 9 shows an example plan for blocks world and the respective simulation in PDSim.

```

(load-truck o23 t2 p2)
(load-truck o1 t2 p2)
(drive t2 p2 ap2 c2)
...
(fly ar1 ap2 ap1)
...

```

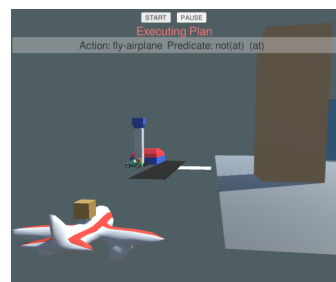
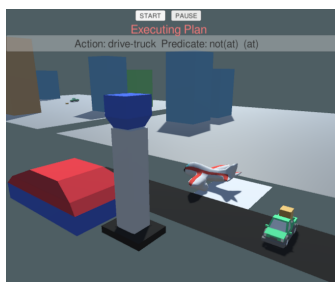


Figure 10: Problem representation for Logistics

```

(move p p5-5 p5-4 up)
(move p p5-4 p5-3 up)
...
(push-nongoal p [ . . ])
...
(push-goal p [ . . ])

```

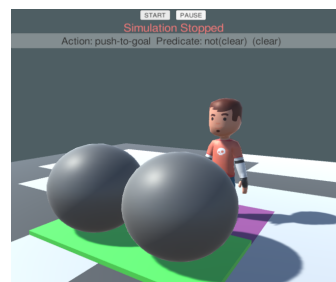
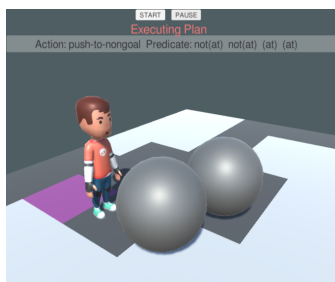


Figure 11: Problem representation for Sokoban

4.2. Logistic

The Logistics (IPC 2000) domain describes a problem involving packages that need to be transported between cities using a plane and within cities using trucks. Constraints of this domain are that each city has one truck and one airport. This domain steps up the complexity of the simulation environment while keeping simple definitions of predicates and actions. *InCity*, *At*, *In* are predicates used to respectively describe if a location is inside a particular city, if an object is in a particular location, and if a package is in a vehicle. Figure 10 shows the plan simulation of the logistic domain, highlighting the translation of boxes between cities using the vehicles.

4.3. Sokoban

The Sokoban (IPC 2008) domain describes the Sokoban game problem¹⁰. The player needs to perform optimal moves to move an object over a prefixed goal on a grid map. Figure 11 illustrates a typical problem level for the Sokoban game where *P* is the player, *S* the stone that needs to be moved, *G* the goal. This domain adds up the complexity from the previous and it was selected to assess the functionality and ubiquity of this simulation program as an additional instrument for the Unity game engine as a tool to rapidly have an AI agent in-game.

```
(move_robot r2 a f)
(pick_up_box r2 box1 f)
(move_robot r2 f a)
(drop_box r2 box1 a)
(move_robot r1 d c)
...
```

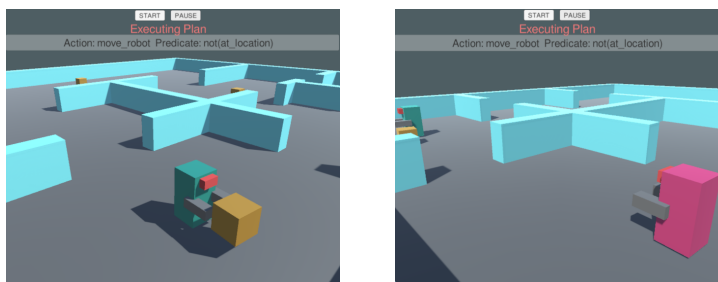


Figure 12: Problem representation for Robots and Boxes

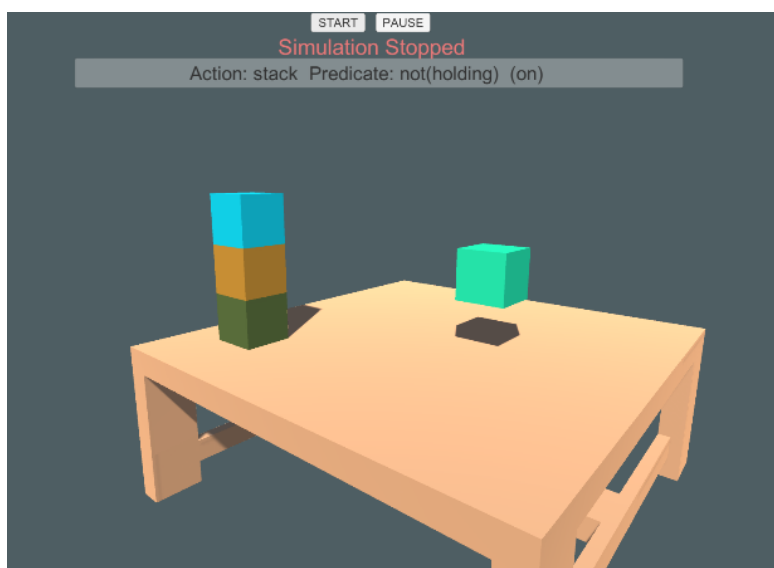


Figure 13: Problem representation for Robots and Boxes

4.4. Robots and Boxes

Finally, the Robots and Boxes domain describes a simple domain where multiple robots can be used to move boxes around a map with rooms and connections between them. The robots can move from a room to another (if those are connected) pick-up or drop boxes. This domain was selected to show that PDSim can be used with custom domain definitions and to demonstrate that can work with Unity's internal components, in particular, the path planning system that it's used to find a smooth path for the robot to follow when moving between rooms. Figure 12 show the simulation for this domain from the point of view of the moving robots.

4.5. Analysis

The planning problems used as a testing ground for PDSim produced a good quality of the simulation in terms of visual cues to evaluate the correctness of a plan. In particular, using the blocks world domain as an example, if we introduce a logic error in the stack action effects plan is generated but it's not valid. The error introduced was about missing to mark as not clear the block below the held block. Figure 13 shows the invalid plan where a block (D) is stuck in mid-air which might indicate an error in the domain modelling.

5. Conclusion

This paper presented the structure and operation of PDSim, a simulation system for PDDL that can be used to animate classical plans. This project supports classical automated planning, however, current work is extending PDSim to support temporal planning through an intuitive visualization system for timed actions and deadlines. Future work on the project will consider support for partial plans defined by the user and other simulation features such as following simulation actions, replaying previous actions, modifying the simulation speed, and displaying partial animations to show the outcome of animation while defining the structure. An important 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.

References

- [1] D. McDermott, M. Ghallab, A. Howe, C. Knoblock, A. Ram, M. Veloso, D. Weld, D. Wilkins, PDDL—the planning domain definition language, Technical Report CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control, 1998.
- [2] R. Howey, D. Long, 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, 2003.
- [3] S. Bensalem, K. Havelund, A. Orlandini, Verification and validation meet planning and scheduling, International Journal on Software Tools for Technology Transfer 16 (2014) 1–12.
- [4] A. Cimatti, A. Micheli, M. Roveri, Validating domains and plans for temporal planning via encoding into infinite-state linear temporal logic, in: Proceedings of AAI, 2017, pp. 3547–3554.
- [5] A. Hill, E. Komendantskaya, R. P. A. Petrick, Proof-carrying plans: A resource logic for ai planning, in: International Symposium on Principles and Practice of Declarative Programming (PPDP), 2020, pp. 1–13.
- [6] M. Fox, D. Long, D. Magazzeni, Explainable planning, in: Proceedings of the IJCAI Workshop on Explainable AI, 2017.

⁹<https://github.com/potassco/pddl-instances>

¹⁰Sokoban Game: <https://en.wikipedia.org/wiki/Sokoban>

- [7] E. De Pellegrin, Pdsim: Planning domain simulation with the unity game engine, in: Proceedings of the ICAPS Workshop on Knowledge Engineering for Planning and Scheduling (KEPS), 2020.
- [8] D. Vrakas, I. Vlahavas, A visualization environment for planning, *International Journal of Artificial Intelligence Tools* 14 (2005) 975–998.
- [9] T. S. Vaquero, V. Romero, F. Tonidandel, J. R. Silva, itsimple2.0: An integrated tool for designing planning domains, in: Proceedings of ICAPS, 2007, pp. 336–343.
- [10] G. Chen, Y. Ding, H. Edwards, C. H. Chau, S. Hou, G. Johnson, M. Sharukh Syed, H. Tang, Y. Wu, Y. Yan, T. Gil, L. Nir, Planimation, 2020. URL: <https://doi.org/10.5281/zenodo.3773027>. doi:10.5281/zenodo.3773027.
- [11] C. Tapia, P. San Segundo, J. Artieda, A PDDL-based simulation system, in: Proceedings of the IADIS International Conference Intelligent Systems and Agents, 2015.
- [12] P. Le Bras, Y. Carreno, A. Lindsay, R. P. A. Petrick, M. J. Chantler, 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), 2020.
- [13] Unity Technologies, Unity, 2021. URL: <https://unity.com>.
- [14] M. Quigley, K. Conley, B. P. Gerkey, J. Faust, ROS: an open-source robot operating system, in: Proceedings of the ICRA Workshop on Open Source Software, 2009.
- [15] R. M. Simpson, D. E. Kitchin, T. L. McCluskey, Planning domain definition using GIPO, *The Knowledge Engineering Review* 22 (2007) 117–134.
- [16] J. Vodrázka, L. Chrupa, Visual design of planning domains, in: Proceedings of ICAPS Workshop on Knowledge Engineering for Planning and Scheduling (KEPS), 2010, pp. 68–69.
- [17] M. C. Magnaguagno, R. Fraga Pereira, M. D. Móre, F. R. Meneguzzi, 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), 2017.
- [18] C. Muise, Planning.domains, ICAPS System Demonstration, 2016.
- [19] B. Bach, C. Shi, N. Heulot, T. Madhyastha, T. Grabowski, P. Dragicevic, Time curves: Folding time to visualize patterns of temporal evolution in data, *IEEE transactions on visualization and computer graphics* 22 (2015) 559–568.
- [20] G. Echeverria, N. Lassabe, A. Degroote, S. Lemaignan, Modular open robots simulation engine: Morse, in: 2011 IEEE International Conference on Robotics and Automation, IEEE, 2011, pp. 46–51.
- [21] O. Gannoni, R. Mukundan, A framework for visually realistic multi-robot simulation in natural environment, arXiv preprint arXiv:1708.01938 (2017).
- [22] J. K. Haas, A history of the Unity game engine, 2014.
- [23] J. Craighead, J. Burke, R. Murphy, 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), 2008.
- [24] M. Helmert, The fast downward planning system, *Journal of Artificial Intelligence Research* 26 (2006) 191–246.