Generating video game puzzles through planning

Generating video game puzzles through planning

Lorenzo James
Industrial Engineering and Innovation Sciences, Information Systems
Eindhoven University of Technology
Eindhoven, The Netherlands
L.J.James@tue.nl

Abstract

Planning is an AI concept that is well-known in the video game industry and has been applied to video games. A large number of these applications have focused on using planning to control the behaviours of agents within video games. However, there is comparatively little research about the application of planning in video games for non-behavioural AI; that is the focus of this work. Many video games consist of puzzles or puzzle elements that players have to solve. Puzzles have a defined search space that can often be considered a planning domain, and planners provide a useful tool for finding a valid series of actions that can efficiently solve such problems. The aim of this work is to explore whether it is possible to create valid and challenge-appropriate puzzles for players in video games through automated planning.

Introduction

The most common use of AI in video games is to to develop behaviors for controlling reactive systems and agents, often coded using techniques like Behavior Trees and Finite State Machines (Yannakakis and Togelius, 2018). For the majority of games these approaches are sufficient; however, as the number of different behaviors for these agents increases, the combination of possible interactions and behaviors can become unmanageable using these techniques (Orkin, 2006). Another downside to using reactive agents is that they cannot be used to create long-term action plans and they have limited to no coordination of interactions with other agents. In contrast, automated planning allows agents to look further ahead in time and implement appropriate strategies to behaviors that apply to the current context. This makes it possible to overcome some of the drawbacks of purely reactive agents. For these reasons, there has been an increased interest in using planning approaches within video games in recent years (Neufeld et al., 2017).

Planners excel at finding a valid series of actions that efficiently achieve a goal-directed problem within a domain. While a large amount of the published research and applications of planning within video games have focused on using planning to control the behaviors of agents within games, there is a lack of research when it when it comes to using planning for non-behavioral AI within games (Neufeld et al., 2017). An interesting non-behavioral element within video games that can be tackled by planning are puzzles, because they are problems that can be solved by logically piecing together elements (Schell, 2008). Information given by the puzzle leaves clues to the player as to which elements need to be manipulated in order to solve the puzzle. These elements can be manipulated by the actions permitted within the game rules. Once the right combination of actions is found, the actions need to be successfully performed by the player to solve the puzzle. In this sense, problems that are tackled by automated planning are similar to puzzles, as solving these problems through planning also requires the right sequence of actions to be found by the planner. Generating a puzzle within a specific game context is a puzzle in itself. This is because puzzles exist within a game level and the difficulty of levels generally increases as players progress through the game. The increase in difficulty is often aligned with keeping players engaged by matching their current skill with an appropriate level of difficulty (Schell, 2008).

This paper explores the possibility of using planning to procedurally generate unique puzzles of an appropriate difficulty within video game levels. As part of this work, a system prototype that generates puzzles within video games using a planner has been developed. In the remainder of the paper, we will detail related work that describes planning for non-behavioral AI in games and puzzle generation with non-traditional AI concepts in video games. We will then outline our approach for using automated planning for puzzle generation in a game environment and describe how our approach is currently implemented. Lastly we will conclude by summing up the current system and describing future work that will be done to the system.

Related work

Prior work on planning for non-behavioral AI in games has been done, mostly in the area of video game narrative which overlaps with the field of Interactive Storytelling (IS), where planning has become a prominent technology for generating IS (Porteous et al., 2010). Planning can be used to generate a narrative and adapt it dynamically, based on the action of an interactor within an IS. This can be done by building a narra-

tive incrementally, decomposing it into subproblems and assigning each of these subproblems a goal. User interactions change the domain, which then causes the planner to re-plan and in turn influences how the next iteration of the narrative is generated. Player Specific Automated Storytelling (PAST) is an AI experience manager that similarly generates narrative based on player actions within video games, and keeps track of the actions players have performed (Ramirez and Bulitko, 2014). If an action results in the narrative going in a direction the author did not intend, the planner generates an alternative narrative that aligns with the author's intention (Ramirez and Bulitko, 2014). Generating narrative through Non-Player Character (NPC)-level planning is also possible. NPCs are the characters within the video game world that the player cannot play as. NPCs can individually use planning to generate their own narrative segments outside of the original story of the game, for instance, by using actions to change the minds of other NPCs in an attempt to change the story to align more with their own goals. This allows NPCs to generate a narrative within the game's universe (Chang, 2009).

Planning can also be used as a tool to help the game design process. This possibility has been explored by modeling the actions within a game, in a domain, and allowing a planner to reason about how these actions can effect the gameplay of the overall design (Zook, 2016). This system allows the planner to create new game mechanics that can be used to tune the design of the game and its gameplay loop (Sicart, 2008).

Our work focuses on the task of procedurally generating puzzles within video games, which is not a new idea: there are a plethora of algorithms that are capable of this task, such as the Breadth First Search algorithm, which can be used to create mazes, and the Monte Carlo Tree Search algorithm, which can be used to generate Sokoban game puzzles (Kartal et al., 2016). Many AI concepts have been used to explore the generation of puzzles such as using evolutionary algorithms to create puzzles with a specific difficulty (Ashlock, 2010) and genetic algorithms to generate Shinro puzzles through natural selection (Oranchak, 2010).

The novel contribution of this project is using the concept of planning to generate puzzles. To the best of our knowledge, there is little prior research in this area. Since planners have the ability to look further ahead in time and plan appropriate strategies, using planning may lead to interesting puzzles. Planners can take into account the current state of the game world and the player. This may be the biggest benefit of using planning to generate puzzles, as this may be able to tailor the generated puzzles to the player and game world circumstances.

Puzzle Game Description

A custom puzzle game prototype has been developed for this project, intentionally designed with the use of planning in mind. Within the game, an environment with multiple puzzles which the player can explore is generated. The goal of the game is for the player to secure the keys, which are each trapped behind the generated puzzles. The player can secure a key once the puzzle is solved.

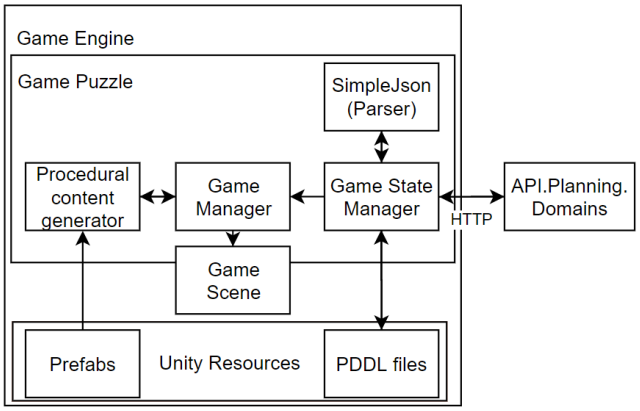

Figure 1: Current implementation of the proposed system

Players have three ability types: Jump, Shoot and Grab. There are four variations of each ability type that can be assigned to the player. The planner generates a puzzle based on the ability types assigned to the player. Puzzles are made out of multiple obstacles, obstacles have the same four variations as player abilities, if an equipped player ability type matches it then the player is able to overcome this obstacle. A puzzle is considered solved when the player has successfully overcome every obstacle within it.

System Overview

The architecture of the system consists of three parts: the puzzle game prototype, the Unity3D game engine and the api.planner.domains planner, as seen in Figure 1. The design of the system aims for high cohesion and low coupling, with the intention of possibly replacing any of the parts in future work.

The puzzle game provides the context and information which the planner uses to generate a plan that will result in a puzzle being generated. The prototype is being developed using the Unity3D game engine, as it provides many standard features found in game engines, such as a rendering engine, physics engine, scene graph and a file management system. Using a game engine cuts down development time and allows for rapid prototyping.

The current prototype consists of multiple levels, the levels describe how many puzzles need to be generated and the difficulty of each one. The levels provide all the information needed for the problem description of the planner. The state manager takes this description and parses it to a PDDL problem file, which is saved in the resources folder in Unity3D and the game manager executes the plan to generate the puzzle in the game world.

The game manager reads both the PDDL problem and domain files from the resources folder, then loads the information into the game, and sends an HTTP Post request to the planner that contains the PDDL problem and domain files. Once the planner has generated a plan, it returns a JSON file containing a PDDL plan to the game manager, as seen in Figure 2. Using SimpleJSON, this file is parsed into the Action class, which holds a list of Action types. All

```
"plan": [
 {
   "action": "(:action create_room
   :parameters (room1)
   :precondition (and
   (not "name": "(create_room room1)")
   )
 },
]
```

Figure 2: A generated plan in plain JSON which needs to be parsed by the JSON parser

instances of the Action class have a name and a list of parameters. When parsed, the name and parameters from each action in the PDDL plan are mapped into an instance of the Action class and are placed into the Actions Queue in the game manager class.

Additionally, the game manager holds a collection of IEnumerators with names that match the actions found in the PDDL Domain file. An IEnumerator is a generic collection class in .NET[1] that can be iterated through. IEnumerators use threading and have the ability to temporarily stop the process of the main system thread until the IEnumerator has fully been executed, so that tasks are executed one at a time. When executing the plan, the game manager loops through the queue of actions and executes the IEnumerator with the matching name of the current action which is being executed. Once the game manager has looped through all the actions in the Action queue, the environment should be generated with the puzzles described by the plan.

Listing 1: Description of a PDDL action for creating a room

```
(:action create_Room
    :parameters(?location - location)
    :precondition
        (and
        (not (initialized ?location))
        )
    :effect
        (and
        (initialized ?location)
        )
)
```

In the current implementation of the system, the IEnumerators are used as functions and are hardcoded to match the names of the actions within the PDDL problem file. The functionality of the IEnumerator is also hardcoded to match the description of the PDDL action; the IEnumerator is matching, as can be seen in Listing 1.

[1]https://docs.microsoft.com/en-us/dotnet/api/system.collections.ienumerator?view=netcore-3.1

When executing an action, the game manager calls the procedural content generator to create a random object of a type defined by the plan. In Listing 2, the IEnumerator calls the Procedural Content Generator to create a random Object of the type room at the location specified within the parameters of the action.

Listing 2: The IEnumerator create_room which matches the PDDL action of the same name found in the domain file

```
IEnumerator
    create_room(List<string>parameters)
{
    string locationName = parameters[0];
    contentGenerator.GetRandomObjectOfType(
    ObjectTypes.Room, locationName);

    yield return null;
}
```

Depending on the difficulty, there are multiple rooms generated within a level. The generated rooms consist of multiple prefab game assets and a puzzle. A puzzle consists of multiple obstacles which in turn also consist of smaller prefabs. Prefabs in Unity3D are meshes that are saved together as a bundle. Some of these prefabs have scripts attached to them to make them interactable with the player. The prefabs that make up a room form choke points to restrict the player from navigating out of the generated rooms and point players towards the puzzles.

Puzzles and rooms are semi-procedurally generated through the procedural content generator and the actions of the game manager. The planner plans which types of prefabs have to be placed within the room or puzzle. The function that executes the plan and places these prefabs, calls the procedural content generator. The procedural content generator then picks a random prefab of the specific type needed and places it randomly on one of the predefined locations within the room or puzzle. Puzzles consist of multiple obstacles that the player has to overcome in order to solve them. Each obstacle is a miniature puzzle of its own and is independent of other obstacles. The obstacles also require the player to have a specific type of ability to solve them. The procedural content generator chooses a random variation of the type needed and places that into the world. When the planner creates a plan to generate a level with puzzles within it, it takes into account the current abilities of the player. The difficulty of a level is determined by the amount of abilities the player needs to use to overcome it and the amount of variety of obstacle types found within the level.

Api.planning.domains was chosen as the planner for this system, because it can act as an independent external system through its API. The API allows PDDL problem and domain files to be sent over HTTP POST requests. Once the files are sent, the planner handles the planning, and if a successful plan is found, the generated plan is sent back in a JSON file which is then parsed by the PDDL parser.

The PDDL problem file used in this project defines all the objects that will be instantiated within the game. Currently, it is also the only PDDL file which gets modified by the system. When there are changes made in the state man-

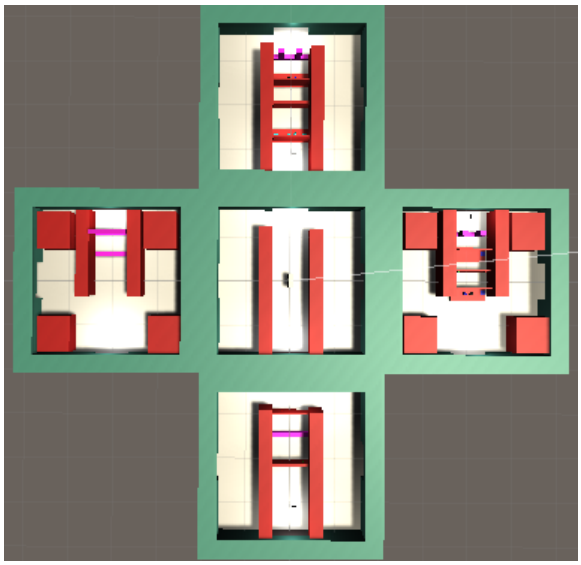

Figure 3: A generated level with 5 rooms that are placed adjacent to each other. Here there are two difficult rooms, one medium room and one easy room. The room in the middle is where the player and locked door are placed.

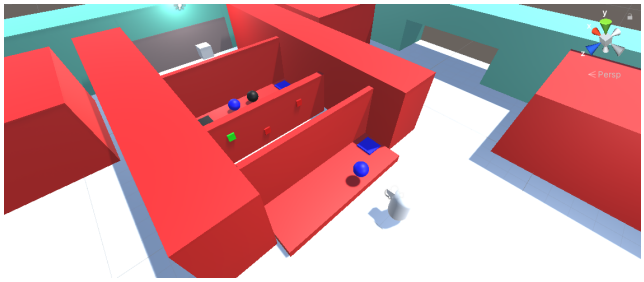

Figure 4: A player attempting to solve one of the three generated puzzles, within a room generated by the planner.

ager script, these changes will be reflected in the problem file, as the initial states of the predicates within the problem file need to represent the current state of the state manager. The state manager decides which and how many rooms need to be generated, it determines the difficulty of each of those rooms, and it also sets the goal of the planner. Before the problem file is sent to the planner, the state manager updates the problem file with the current state of the state manager. Currently, there are some limitations in what can be modified in the problem file. In the (:Init) function of PDDL, which defines the initial state of the world, there must always be at least one room present since the system is not prepared to handle scenarios in which there are no puzzles or rooms to be generated. Presently, there must always be one room generated, which is done by the following PDDL code: (Difficulty easy room2). Another limitation is that the first goal set in the (:goal) function of PDDL needs to be (playerset player). Otherwise, the player will never be placed in the game world.

This project makes use of PDDL typing, which allows PDDL to assign a type to each object within the problem file. This is used to force certain parameters of PDDL actions found in the domain file to be of a specific type. The following are all the types that are defined and used in the problem file of this project:

- **Element** types are elements in the game world which are tied to the win/lose conditions in the game. (Objects that fall under elements are doors and keys.)
- **Location** types are prefab rooms that up the level where the elements, obstacles, and player can be placed.
- **Obstacle** types are the type of puzzle prefabs that can be placed within a level.
- **Level** type represents the difficulty level of a level.
- **Player** type represents a playable character.

The domain file used in this project goes largely unchanged; it describes the actions that the planner searches through to reach the goal state described in the problem file. It also describes the IEnumerators that are used in the game manager class. The predicates in the domain file also use PDDL typing.

Currently, the system generates puzzles by checking the player's current abilities, and then based on that, it spawns a predefined puzzle piece that can be solved by using the player's abilities as found by the system. The amount of puzzle pieces that are spawned are based on the level of difficulty that is set. In the current implementation, puzzles are only semi-generated by the system, as they are built by piecing together level pieces. In future work, the design of the system will be updated so that the system is able to generate level pieces based on the player's current abilities and the difficulty of the level. Each ability type has a standard puzzle to begin with, and based on the difficulty of the level to which the puzzle is added, this standard puzzle will be modified. The goal state of the planner will be based on the modified puzzle. Actions of the planner will be assigned a difficulty, through an equivalent of PDDL typing. The difficulty of the puzzle will determine the amount of easy, medium or hard actions it should take for the puzzle to be completed. This redesign is also being considered in order to make it possible for the system to fully generate puzzles. Not only would the level pieces be generated, but the whole puzzle would be fully generated through this redesigned system. Future iterations of the systems will be tested by users, to generate feedback on the practicality and entertainment level of the generated puzzles. In future study, a more classical implementation of level generation will also be implemented and bench-marked against the proposed system, in order to compare if planning can indeed handle generating larger and more complex puzzles better than traditional techniques.

Conclusion and Future Work

The work described in this paper was developed as part of a larger project aimed at exploring the possibility of using planning to procedurally generate unique puzzles within video games. Currently, the system is made up of three components: prototype of a game, Unity3D Game Engine, and

the API.Planning.Domains planner. The puzzles in the game prototype are a collection of prefabs. This system is capable of semi-procedurally generating puzzles within the level. The puzzles are generated by following the plan defined by the planner, and the game manager parses and executes this plan. The actions that are executed make use of the procedural content generator to generate parts of the obstacles which make up the puzzles within the game. By using a planner, the system considers all the elements needed to both solve and generate the puzzle, and creates a plan based on that. The state of the game world is reflected in the problem file, when the planner fails to generate a plan based on the current game state, the planner is equipped to change elements in the game world. The changing of said elements, changes the game state into a state that makes it possible to generate a plan. Currently, the planner is only able to manipulate the game world, by adding abilities to the current game state, which are needed for players to be able to solve the generated puzzle. The manipulation of the game world can be extended to other game elements, allowing the system to possibly dynamically alter the game in interesting ways.

The current design has limitations to it, such as the lack of support for real-time planning and re-planning of goals, the lack of ability to create plans to fully generate puzzles and obstacles, and the lack of ability to truly randomize the position of the generated puzzles and objects. Furthermore, the current setup of the game manager actions and the domain file actions are highly coupled and hardcoded.

In future work, the project may be expanded by using the planner to also generate obstacles themselves. This would result in the system being less reliant on randomly choosing from prefabs to build puzzles and it would make it a more procedural system. Another facet to be added is real-time puzzle generation. In order to achieve real-time puzzle generation, the system would need to be able to dynamically change the goals of the domain files and have the ability to re-plan. Future implementations include redesigning the system to decouple the actions of the game manager and the actions in the domain file. The planner will also be replaced by a planner built into the Unity3D engine to improve planning time. Overall, the system prototype is promising and with more iterations of the system and abstractions of the plans, we aim to improve the system to result in more fully procedurally generated puzzles.

Acknowlegments

This work has been done as part of an Artificial Intelligence MSc dissertation project at Heriot-Watt University, under the guidance of Dr. Ron Petrick.

References

Ashlock, D. (2010). Automatic generation of game elements via evolution. In *Proceedings of the 2010 IEEE Conference on Computational Intelligence and Games*, pages 289–296. IEEE.

Kartal, B., Sohre, N., and Guy, S. (2016). Generating sokoban puzzle game levels with monte carlo tree search. In *The IJCAI-16 Workshop on General Game Playing*, page 47.

Neufeld, X., Mostaghim, S., Sancho-Pradel, D., and Brand, S. (2017). Building a planner: A survey of planning systems used in commercial video games. *IEEE Transactions on Games*.

Oranchak, D. (2010). Evolutionary algorithm for generation of entertaining shinro logic puzzles. In *European Conference on the Applications of Evolutionary Computation*, pages 181–190. Springer.

Orkin, J. (2006). Three states and a plan: the ai of fear. In *Game Developers Conference*, volume 2006, page 4.

Porteous, J., Cavazza, M., and Charles, F. (2010). Applying planning to interactive storytelling: Narrative control using state constraints. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 1(2):1–21.

Ramirez, A. and Bulitko, V. (2014). Automated planning and player modeling for interactive storytelling. *IEEE Transactions on Computational Intelligence and AI in Games*, 7(4):375–386.

Schell, J. (2008). *The Art of Game Design: A book of lenses*. CRC press.

Sicart, M. (2008). Defining game mechanics. *Game Studies*, 8(2):n.

Yannakakis, G. N. and Togelius, J. (2018). *Artificial intelligence and games*, volume 2. Springer.

Zook, A. (2016). *Automated iterative game design*. PhD thesis, Georgia Institute of Technology.