Automated Gameplay Generation from Declarative World Representations

Justus Robertson and **R. Michael Young**
Liquid Narrative Group
Department of Computer Science
North Carolina State University
Raleigh, NC 27695
jjrobert@ncsu.edu, young@csc.ncsu.edu

Abstract

An open area of research for AI in games is how to provide unique gameplay experiences that present specialized game content to users based on their preferences, in-game actions, or the system's goals. The area of procedural content generation (PCG) focuses on creating or modifying game worlds, assets, and mechanics to generate tailored or personalized game experiences. Similarly, the area of interactive narrative (IN) focuses on creating or modifying story worlds, events, and domains to generate tailored or personalized story experiences. In this paper we describe a game engine that utilizes a PCG pipeline to generate and control a range of gameplay experiences from an underlying IN experience management construct.

Introduction

Procedural content generation has played a large role in the video game industry since its early days (e.g. *Rogue* (Toy and Wichman 1980), *Diablo* (Blizzard Entertainment 1996), *Dwarf Fortress* (Bay 12 Games 2006), *Borderlands* (Gearbox 2009), and *Galactic Arms Race* (Hastings, Guha, and Stanley 2009)). Recently, *Minecraft* (Mojang 2011) has become immensely popular in part because of its expansive worlds procedurally generated from one of 2^{64} numerical seeds. In academia, researchers are working to combine the power of content generation with models from AI and human-computer interaction to create algorithms that not only create varied experiences, but experiences custom-tailored to specific users (Yannakakis and Togelius 2011).

Similarly, branching story games like the *Choose Your Own Adventure* series (Packard 1979) have existed and informed game design since its early years. Recent games like *Mass Effect* (BioWare 2007) and *The Walking Dead* (Telltale Games 2012) have gained popularity for incorporating user-driven story decisions which affect game events. In academia, researchers are working to procedurally generate interesting stories that branch and change according to user choices without the need for a human author to hand-craft each path (Riedl and Bulitko 2013). When completed, this research will create a system that build long chains of plot events intertwined with and dependent on the user's actions with rich narrative properties to be created by an interactive story telling agent, called an *experience manager*.

The combined vision of procedural content generation and interactive narrative is best exemplified by the holodeck from *Star Trek: The Next Generation*. The holodeck is a fully-immersive 3D environment generated by computers that can dynamically create and revise world mechanics, assets, states, virtual agents, and plot events. The Enterprise's computer acts as an experience manager by taking as input voice commands or programmatic instructions that describe a desired experience. Using the content generation modules at its disposal, the system then generates a story world from the input description and guides the user's interactions within it. One of the strengths of the holodeck is its ability to function as a hub for different technologies that create a single, unified experience.

In this paper we describe a similar hub for connecting modules that generate world mechanics, assets, states, virtual agents, and plot events that allows an experience manager direct access to the game world in which the player interacts. This system, called the General Mediation Engine (GME), functions by layering a procedural content generation pipeline on top of an experience management framework. This PCG pipeline takes as input sets of atomic formulae maintained by the experience manager that represent world states and produces an interactive game by instantiating, controlling, and destroying assets based on the state.

Related Work

GME is an experience management and content generation pipeline that allows a game world and its dynamics to be generated from an asset library and an experience manager's declarative state model. Our declarative state model is defined with PDDL (McDermott et al. 1998), a formal language for specifying planning problems. PDDL representations consist of an initial state description, a goal state, and a set of action operators that can be performed by agents to transform world states based on a set of preconditions, conditions that must hold true in the current state, and effects, conditions that become true once the action has executed. In our approach, both preconditions and effects are expressed as logical atomic formulae.

From the PDDL input we construct a state transition system that allows the player and other agents to input

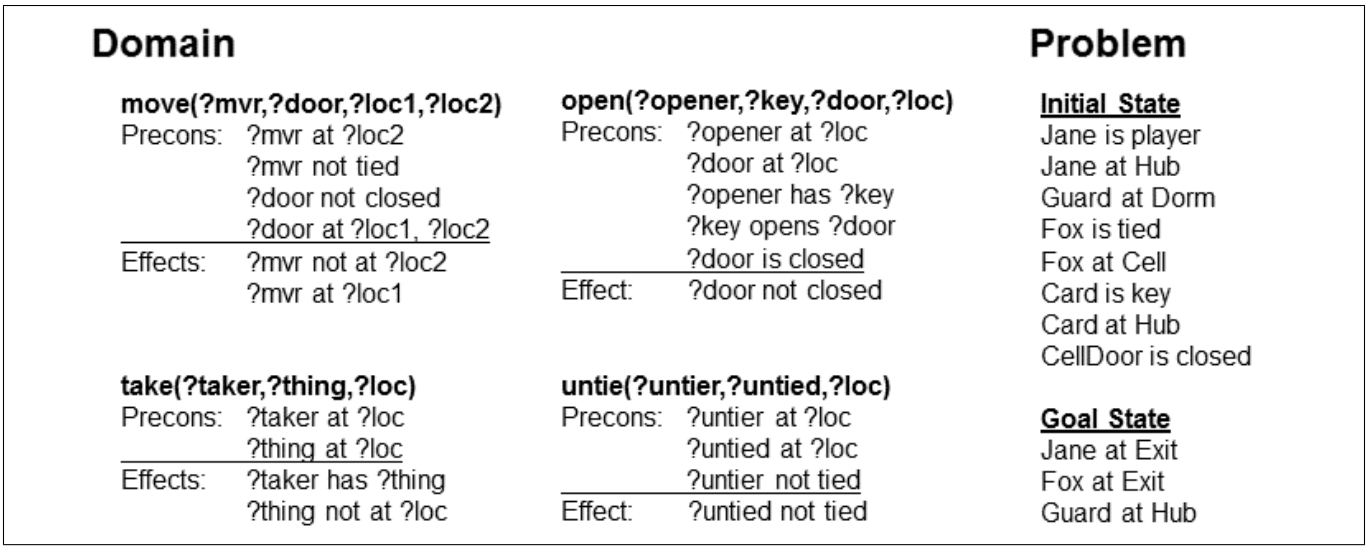

Figure 1: A simplified, informal description of the Base Case domain and problem. In this figure, preconditions and effects are described in english-like syntax for readability. Tokens with question marks as their initial character (e.g., ?loc) are variables. Words that begin with upper-case letters (e.g., Jane) are object constants. This domain includes action descriptions for moving around a military base, picking up objects, unlocking doors with keys and untying prisoners. The planning problem shown here involves Jane rescuing Fox, who is held prisoner in the base.

actions whose preconditions are true in the current state. This state transition system is similar to the game used by PAST (Ramirez and Bulitko 2014), an experience manager that reasons about player preferences, in its Little Red Riding Hood evaluation (Sturtevant et al. 2014). This central state transition system is connected to a planner capable of finding paths of actions from the current to the goal state. The system currently supports versions of Fast Downward (Helmert 2006), an efficient off-the-shelf planner successful in planning competitions, and Glaive (Ware and Young 2014), a specialized narrative planner that reasons about intention and conflict, but any system capable of producing solutions to PDDL problems can be used.

The system's underlying experience manager draws from a tradition of plan-based methods, including Mimesis (Young et al. 2004), ASD (Riedl et al. 2008), and the system used by The Merchant of Venice (Porteous, Cavazza, and Charles 2010). These systems sense and effect their game environment according to a narrative plan and update or generate new plans based on user interaction. Riedl (2005) gives an overview of how ASD and similar systems interface with commercial game engines to create playable experiences.

In contrast, our system generates and updates a game world layout from the declarative AI formalism given in the initial PDDL state. This is similar to other generative systems that use a formal model to create game spaces. Launchpad (Smith et al. 2011) is a grammar-based approach to 2D platforming level generation that pieces together common elements of the genre into levels, Game Forge (Hartsook et al. 2011) automatically creates a game world to support a given story plan by using a genetic algorithm to generate a world graph according competing design metrics, and Valls-Vargas, Ontanón, and Zhu (2013) describe a system that generates world maps which support different arrangements of plot events and evaluates the world and story configurations.

In addition to an initial world representation, our system uses a declarative formalism to generate state dynamics. Other declarative approaches have successfully modeled, analyzed, and generated game mechanics. LUDOCORE (Smith, Nelson, and Mateas 2010) is a game engine built on event calculus (Mueller 2004) that generates gameplay traces using its logical representation of a game world and mechanics, Zook and Riedl (2014) describe a system for generating game mechanics using AI planning action representations and Answer Set Programming that is shown to generate avatar-based game mechanics for an RPG and 2D platformer, and *MKULTRA* (Horswill 2014) is a detective game that implements a SHRDLU-like (Winograd 1972) NPC in Prolog that can interact with the player and answer questions about its environment.

System Overview

GME is a game engine that, given a formal world state and transition description in PDDL, produces a state transition system. This system allows a human player and virtual characters to iteratively take action and update the world according to state transition operators described in the formal model. This transition system serves as a central world model for the system's *experience manager* and *discourse generator*. The experience manager is a system that maintains a desired experience plan, monitors the transition system, executes NPC character actions from the plan, and manipulates the world directly as the player interacts with the game. The discourse manager monitors the transition system and generates a playable game world for the user to interact with based on the underlying state description and a library

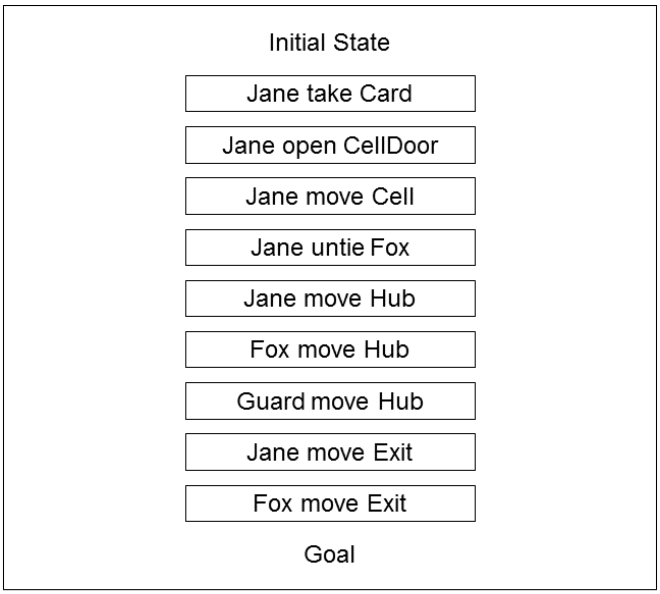

Figure 2: A possible plan for the initial Base Case problem.

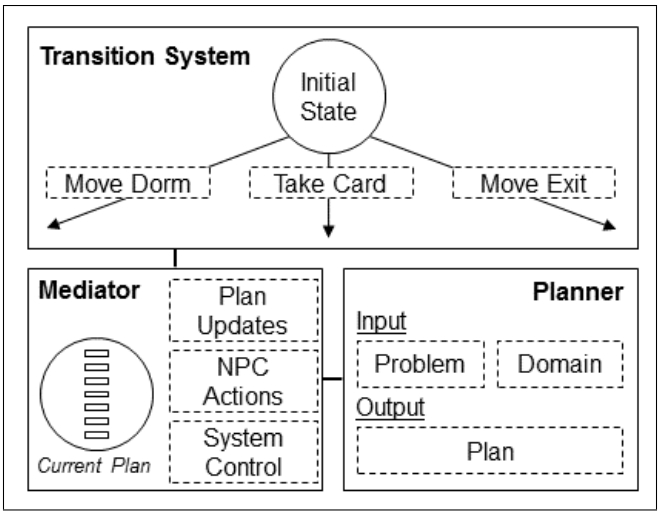

Figure 3: The three modules that comprise GME's experience management framework: a state transition system, a planner, and a mediator that connects and controls the modules.

of art, animation, and code assets. The discourse manager relays player actions from the game model to the transition system, animates player and NPC actions, and manages local NPC behavior outside of plan directives.

The Experience Manager

GME's experience manager is comprised of a *state transition system*, a *planner*, and a *mediator* that work together to maintain world states, update plan information, and execute NPC actions. The architecture is shown in Figure 3 and is implemented in the form of mediation game trees (Robertson and Young 2014b).

State Transition System

GME's state transition model is built from a PDDL description of an initial world state and a domain of operators that specify how the world state can change according to player and NPC actions. Figure 1 is an informal description of an example domain and problem called Base Case that we use as a running example. In order for an action to be performed by an agent in a state, the action's set of preconditions must hold true. If a valid action is performed by an agent in the state transition system, the action's effects will be applied to the current state in order to produce the next world state.

Planner

The system's planner is an external module that takes as input a domain and planning problem, and returns a set of steps that transform the problem's initial state into its goal state. Figure 2 is one possible solution for the initial Base Case problem that may be found and returned by the system's planner at the start of execution.

Mediator

The system's mediator interfaces with the state transition system and external planner to drive the experience by maintaining the current plan, issuing commands to NPCs, and requesting new experience plans when necessary.

Discourse Generation

The narrative theorist Chatman (1980) draws the distinction between the abstract events that take place in a narrative world, which are part of the *story*, and the way these events are arranged and presented to an audience, which is part of a *discourse*. We use this distinction between story and discourse to conceptualize GME's experience manager and procedural content generation pipeline. The abstract, declarative backend that maintains state information and issues high-level commands to game characters is the system's story generator and the game world frontend that is created from the underlying declarative model is the system's discourse generator.

In this section we outline the architecture of a discourse generator intended for 2D avatar-based games. The generator is comprised of six modules that create, configure, and manage the game world based on the underlying declarative transition system and libraries of code, art, and animation assets. The *experience management interface* initializes the discourse generation system, interfaces with the underlying transition system, and relays commands from the mediator. The *game state manager* accepts a game state description from the experience manager and is responsible for instantiating, maintaining, and destroying game assets based on the current state. A *user interface generator* and *level generator* create and configure the interface and world layout at initialization. A *player architecture* exists to allow local behaviors not modeled by the experience manager, sense when a state transition action has been taken, and relay the player's action to the experience manager. Finally, an *NPC architecture* exists to allow local intelligent behavior, receive instructions from the mediator, and animate actions in the game world. Figure 4 is a diagram of the full system pipeline.

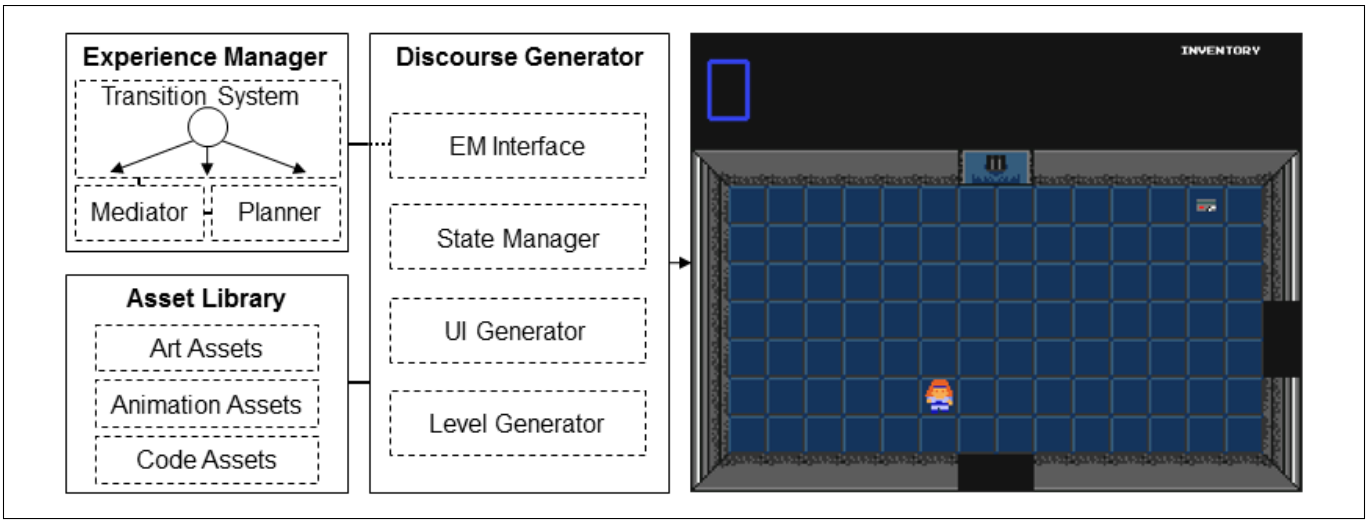

Figure 4: The full system pipeline from declarative representation and game assets, to discourse generator, to game world.

In addition to describing the general components of a discourse generation system we also provide a running example of an implementation of the pipeline shown in Figure 4 using the Unity game engine (Unity Technologies 2005 2015). The implementation is called the Unity General Mediation Engine, or UGME. The experience management framework, implemented in C#, is given to UGME as a compiled DLL. UGME is also provided with a library of prefabs, or object templates that contain specific properties and assets, for possible declarative objects it can instantiate in the game world. Each of these prefabs contains a 32X32 pixel sprite, a finite state machine that controls animations, and code that governs local behavior like pathfinding. The game is initialized and maintained from the experience manager's declarative transition system and a Unity scene that contains the four discourse generation modules. We discuss UGME in the context of the game Base Case, a top-down sneaking game implemented using the input given in Figure 1.

Experience Management Interface

In order to synchronize the interactive game world with the experience manager's underlying declarative state, the discourse manager must be able to access the current declarative world state description, send actions the player takes in the game world to the declarative representation, and send actions prescribed by the experience manager's plan to game world NPCs. The experience management interface fulfills these three roles. It exposes the current declarative state description to the discourse manager, sends fully ground atomic formula that represent player actions to the experience manager, and sends plan steps from the experience manager to game world NPCs for execution.

User Interface Generator

Based on the type of game that is being created, non-game world elements such as user interface displays and camera behavior will change. The user interface generator is responsible for adding UI elements and configuring camera behavior based on special PDDL formulas. For example, the Base Case game shown in Figure 4 has the *Inventory* UI element in addition to the main game display. The inclusion of this UI element is triggered by a special formula, *(inventory)*, in the planning problem's initial state. When the user interface generator finds this formula in the initial state description it decreases the height of the main display, enables a second camera for the inventory display, and instantiates and destroys game objects in the inventory based on the declarative world state.

The user interface generator is also responsible for initializing the game's camera. For example, the Base Case game camera behaves similarly to *The Legend of Zelda*'s (Nintendo 1986), which scrolls whenever the player moves between screen segments. However, a wide range of 2D camera behaviors can be created from knowledge of the player's position and a few other objects (Keren 2015) which can be implemented as a library and chosen with a declarative statement, similar to UI elements.

Level Generator

The discourse generator needs a game world in which the player and NPCs can interact that matches the declarative environment maintained by the experience manager. The level generator is responsible for building this game environment at initialization from special formulas defined in the declarative state. The PDDL planning problems used by the system specify some number of discrete world locations, called *locations*, and navigable edges between these locations, called *connections*. From this set of locations and connections, the level generator builds a high-level graph that realizes a possible physical configuration of the locations given the connections.

From this high-level graph, the level generator creates the realized game world locations as collections of environment game objects. Locations are labeled with a type predicate in the state description that tells the level generator what method to use when instantiating game objects. For exam-

ple, in the Base Case implementation every location corresponds to a room the player can visit. All the rooms are of type *base* which tells the system to use the *base* method defined in the level generator to create the room. The base generator creates a room to be the size of the user's screen, it instantiates specific wall tiles around the room's perimeter, creates door objects on the perimeter between connected locations, and populates the room's interior with special floor tiles that enable pathfinding.

Game State Manager

In addition to generating environments, the discourse generator must create and maintain world objects like the player, NPCs, and items. The game state manager is responsible for instantiating, maintaining, and destroying these game objects based on the current declarative world state. For every location in the world state, the game state manager stores a table of the instantiated objects that exist at the location in the game. Whenever the experience manager's declarative state is updated, the game state manager refreshes the instantiated world objects at the player's location. One way the game state manager refreshes the world is by instantiating and destroying game objects.

For example, if the player executes *take(Jane, Card, Hub)* from the initial state by walking to the Card and pressing a button that corresponds to the *take(?taker, ?thing, ?loc)* action, the action will be sent to the experience manager which will produce a new declarative state where *at(Card, Hub)* is no longer true and *has(Jane, Card)* is true. The game state manager will refresh the game world by checking the declarative state of the current room against its table of instantiated game objects. Since the Card no longer exists at the Hub in the declarative state but a game object representing the Card exists in the game state manager's table, it will destroy the Card object from the game world.

The game state manager also updates the properties of instantiated game objects. For example, if the player takes the Card, walks to the Cell Door, and presses a button that corresponds to the *open(?opener, ?key, ?door, ?loc)* action, they will trigger *open(Jane, Card, CellDoor, Hub)* to be executed by the experience manager. This will produce a new declarative state where *closed(CellDoor)* is false.

All objects that may have different properties, such as doors that can be open or closed, have an animation controller. An animation controller is a Unity asset similar to a finite state machine that controls a game object's sprite based on values maintained by the controller. The door's animation controller has a boolean switch called *closed* that if true will display a door sprite and if false will display a black floor tile. Doors also have a script attached to them that allows the object to be navigable by the player and pathfinding NPCs only when *closed* is false.

While refreshing, the game state manager makes sure that all booleans of an object's animation controller are toggled true or false according to the current declarative state. In our example, the Cell Door game object will become open and navigable to the player and NPCs after the game state manager's refresh due to the *closed(CellDoor)* condition becoming false in the declarative state.

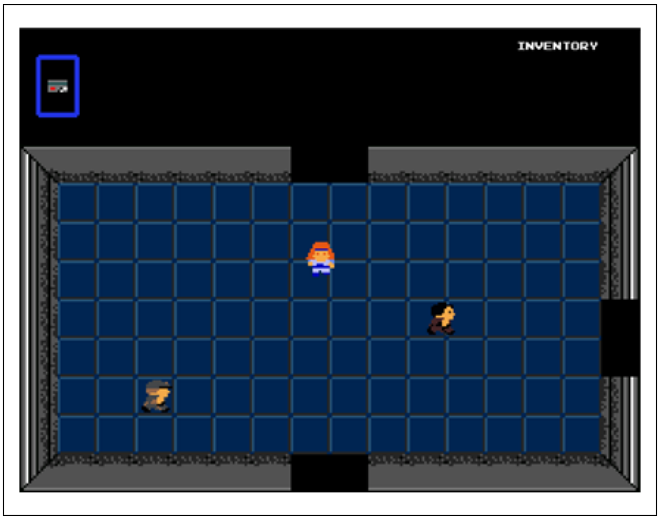

Figure 5: The player and Fox escaping from the Hub while avoiding the Guard, who has moved from the Dorm.

Player Architecture

If the declarative model does not fully account for the game world mechanics, such as the (x,y) position of the player, then local behaviors and animations such as player movement inside the confines of a location must be handled by the discourse generator. The discourse generator must also map local actions, such as pressing an action button when colliding with the Card game object, to declarative action requests, such as the operator *take(Jane, Card, Hub)*. This functionality is handled by scripts attached to the player. These scripts can be attached at initialization from a library based on the initial state description, similar to the user interface system.

NPC Architecture

The discourse generator should allow NPCs to exhibit local behavior but also receive and execute commands issued by the experience manager. To this end, NPCs have a set of baseline behaviors, like pathfinding from one floor tile to another, that can be sequenced to create more complex behaviors like patrolling a room. Local behaviors can be built and repeated for different types of NPCs, like patrolling a room and glancing in different directions for enemy guards. NPCs also maintain a queue of action requests from the experience manager based on the current plan. When an NPC detects one or more queued experience management actions, it uses reflection to invoke a method that corresponds to the action type from a central method library that describes how to execute each type of declarative action.

For example, in the Base Case plan pictured in Figure 2 the Guard will move from the Dorm to the Hub as the seventh action. When this happens, the experience manager interface will send a fully ground action operator object to the Guard NPC's action queue. When the Guard next makes a decision about what action to perform, it will notice that there is an action object, *move(Guard, DormDoor, Hub, Dorm)*, waiting in its action queue. The Guard will then invoke the *move* method in its action library using it-

Figure 6: A demonstration of the computer object that allows the player to toggle conditions in the world true or false. The left picture shows the initial state of the Base Case world with a computer screen overlay with select conditions that are true in the state. The right picture shows the game world after the player has deselected one of the conditions and the door to the cell has opened due to the state manipulation.

self and the action object as parameters. The *move* object in the Base Case NPC action library finds a path between the calling NPC and a randomly selected navigable floor tile that is a child of the game object that shares a name with the *?loc1* term of the action object, which is the Hub. The *move* method adds the returned path to the Guard's pathfinding queue and the Guard moves to the Hub before resuming its local patrol routine. The Guard patrolling the Hub after receiving the move command from the experience manager is shown in Figure 5.

Base Case: Escape a Recursive Prison

In the Base Case game, the player must free prisoners from a military compound while avoiding detection from patrolling guards. Every time the player escapes with a freed prisoner, the world is reconfigured to a PDDL problem file that serves as an initial state for a new version of the game. In order to complete the game and escape the base for good, the player must find and use a special computer hidden in the base. This computer is a special-purpose game object that allows the player to toggle conditions in the underlying world state on and off. The computer works by changing conditions directly in the state manager, which requests for GME to initialize a new state transition system using the modified declarative world state. An example of the computer being used by the player along with the state changes it creates is presented in Figure 6.

Discussion and Future Work

Given the right library of assets, UGME could be used in conjunction with a natural language authoring tool to create game worlds in a manner similar to Inform (Graham Nelson 2014) and other interactive fiction design systems. A game author could create a game world and choose its mechanics using the authoring tool, which would create PDDL files from the commands, which would run UGME.

As planning technology grows more powerful, more of the game's mechanics such as the game-world position of characters, can be modeled in the state transition system instead of as a local behavior in the discourse generator. A method for generating game mechanics automatically from the PDDL world description, similar to that used by Zook and Riedl (2014), might also be used to generate the code that governs the local behavior of players and NPCs.

Another interesting direction for future work is the development of procedural asset generation. Cook and Colton (2014) and Hodhod, Huet, and Riedl (2014) both experiment with querying Sketchup Warehouse, a library of 3D models with standard measurements, to populate a scene with assets. If an asset generation pipeline could be created to generate art and animation assets from a PDDL description it would greatly reduce the authoring burden of maintaining a library of assets.

Experience management strategies such as Robertson and Young (2014a) and Ware and Young (2010) modify aspects of the world to further the audience's experience. New strategies for managing player activity can be integrated into GME's declarative back-end, change conditions in the state transition system on the fly. In this manner, the changes will be automatically integrated into the game world.

Conclusion

The General Mediation Engine is a framework for generating games from a declarative AI world representation using a procedural content generation pipeline. The framework has been implemented in the Unity game engine as the Unity General Mediation Engine. This game engine has been used to create a top-down sneaking game called Base Case that is generated and maintained by a declarative backend.

References

Bay 12 Games. 2006. Dwarf Fortress.

BioWare. 2007. Mass Effect.

Blizzard Entertainment. 1996. Diablo.

Chatman, S. B. 1980. *Story and Discourse: Narrative Structure in Fiction and Film*. Cornell University Press.

Cook, M., and Colton, S. 2014. Ludus Ex Machina: Building A 3D Game Designer That Competes Alongside Humans. In *Proceedings of the 5th International Conference on Computational Creativity*.

Gearbox. 2009. Borderlands.

Graham Nelson. 2014. Inform 7.

Hartsook, K.; Zook, A.; Das, S.; and Riedl, M. O. 2011. Toward Supporting Stories with Procedurally Generated Game Worlds. In *Computational Intelligence and Games*, 297–304. IEEE.

Hastings, E. J.; Guha, R. K.; and Stanley, K. O. 2009. Automatic Content Generation in the Galactic Arms Race Video Game. *Computational Intelligence and AI in Games, IEEE Transactions on* 1(4):245–263.

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

Hodhod, R.; Huet, M.; and Riedl, M. 2014. Toward Generating 3D Games with the Help of Commonsense Knowledge and the Crowd. In *Artificial Intelligence and Interactive Digital Entertainment*.

Horswill, I. 2014. Game Design for Classical AI. In *Tenth Artificial Intelligence and Interactive Digital Entertainment Conference*.

Keren, I. 2015. Scroll Back: The Theory and Practice of Cameras in Side-Scrollers. `http://gamasutra.com/blogs/ItayKeren/20150511/243083/Scroll_Back_The_Theory_and_Practice_of_Cameras_in_SideScrollers.php`. Accessed: May 29, 2015.

McDermott, D.; Ghallab, M.; Howe, A.; Knoblock, C.; Ram, A.; Veloso, M.; Weld, D.; and Wilkins, D. 1998. *PDDL - The Planning Domain Definition Language*.

Mojang. 2011. Minecraft.

Mueller, E. T. 2004. Event Calculus Reasoning Through Satisfiability. *Journal of Logic and Computation* 14(5):703–730.

Nintendo. 1986. The Legend of Zelda.

Packard, E. 1979. *The Cave of Time*. Choose Your Own Adventure. Bantam Books.

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* 1(2):10.

Ramirez, A., and Bulitko, V. 2014. Automated Planning and Player Modelling for Interactive Storytelling. *IEEE Transactions on Computational Intelligence and AI in Games*.

Riedl, M., and Bulitko, V. 2013. Interactive Narrative: An Intelligent Systems Approach. *AI Magazine* 34(1):67–77.

Riedl, M. O.; Stern, A.; Dini, D. M.; and Alderman, J. M. 2008. Dynamic Experience Management in Virtual Worlds for Entertainment, Education, and Training. *International Transactions on Systems Science and Applications* 4(2):23–42.

Riedl, M. O. 2005. Towards Integrating AI Story Controllers and Game Engines: Reconciling World State Representations. In *IJCAI Workshop on Reasoning, Representation and Learning in Computer Games*.

Robertson, J., and Young, R. M. 2014a. Finding Schrödinger's Gun. In *Artificial Intelligence and Interactive Digital Entertainment*.

Robertson, J., and Young, R. M. 2014b. Gameplay as On-Line Mediation Search. In *Experimental AI in Games*.

Smith, G.; Whitehead, J.; Mateas, M.; Treanor, M.; March, J.; and Cha, M. 2011. Launchpad: A Rhythm-Based Level Generator for 2-D Platformers. *Computational Intelligence and AI in Games, IEEE Transactions on* 3(1):1–16.

Smith, A. M.; Nelson, M. J.; and Mateas, M. 2010. Ludocore: A Logical Game Engine for Modeling Videogames. In *Computational Intelligence and Games (CIG), 2010 IEEE Symposium on*, 91–98. IEEE.

Sturtevant, N. R.; Orkin, J.; Zubek, R.; Cook, M.; Ware, S. G.; Stith, C.; Young, R. M.; Wright, P.; Eiserloh, S.; Ramirez-Sanabria, A.; et al. 2014. Playable Experiences at AIIDE 2014. In *Tenth Artificial Intelligence and Interactive Digital Entertainment Conference*.

Telltale Games. 2012. The Walking Dead.

Toy, M., and Wichman, G. 1980. Rogue.

Unity Technologies. 2005-2015. Unity.

Valls-Vargas, J.; Ontanón, S.; and Zhu, J. 2013. Towards Story-Based Content Generation: From Plot-Points to Maps. In *Computational Intelligence in Games (CIG), 2013 IEEE Conference on*, 1–8.

Ware, S. G., and Young, R. M. 2010. Rethinking Traditional Planning Assumptions to Facilitate Narrative Generation. In *AAAI Fall Symposium: Computational Models of Narrative*.

Ware, S. G., and Young, R. M. 2014. Glaive: A State-Space Narrative Planner Supporting Intentionality and Conflict. In *Artificial Intelligence and Interactive Digital Entertainment*.

Winograd, T. 1972. Understanding Natural Language. *Cognitive Psychology* 3(1):1–191.

Yannakakis, G. N., and Togelius, J. 2011. Experience-Driven Procedural Content Generation. *IEEE Transactions on Affective Computing* 2(3):147–161.

Young, R. M.; Riedl, M. O.; Branly, M.; Jhala, A.; Martin, R. J.; and Saretto, C. J. 2004. An Architecture for Integrating Plan-Based Behavior Generation with Interactive Game Environments. *Journal of Game Development* 1(1):51–70.

Zook, A., and Riedl, M. O. 2014. Automatic Game Design via Mechanic Generation. In *Proceedings of the 28th AAAI Conference on Artificial Intelligence*.