Intelligent Narrative Technologies: Papers from the 2012 AIIDE Workshop
AAAI Technical Report WS-12-14

Procedural Game Adaptation: Framing Experience Management as Changing an MDP

David Thue and **Vadim Bulitko**
Department of Computer Science
University of Alberta
Edmonton, AB, T6G 2E8, CANADA
dthue | bulitko @ ualberta.ca

Abstract

In this paper, we present the Procedural Game Adaptation (PGA) framework, a designer-controlled way to change a game's dynamics during end-user play. We formalize a video game as a Markov Decision Process, and frame the problem as maximizing the reward of a given player by modifying the game's transition function. By learning a model of each player to estimate her rewards, PGA managers can change the game's dynamics in a player-informed way. Following a formal definition of the components of the framework, we illustrate its versatility by using it to represent two existing adaptive systems: PaSSAGE, and *Left 4 Dead*'s AI Director.

Introduction

Changing the dynamics of a video game (i.e., how the player's actions affect the game world) is a fundamental tool of video game design. In *Pac-Man*, eating a power pill allows the player to temporarily defeat the ghosts that pursue and threaten her for the vast majority of the game; in *Call of Duty 4*, taking the perk called "Deep Impact" allows the player's bullets to pass through certain walls without being stopped. The parameters of such changes (e.g., how much the ghosts slow down while vulnerable) are usually determined by the game's designers long before its release, with the intent of applying them uniformly to every player's experience (e.g., the ghosts' slower speed depends only on the current level of the game). In cases where the game's dynamics can change in significant ways, it becomes important to choose the parameters of those changes well, for the game's public reception may depend crucially on the appeal of the gameplay that results (e.g., *StarCraft* has benefited from Blizzard's significant effort spent refining its game dynamics). Furthermore, if certain changes to the game's dynamics are contentious, their uniform application might fail to satisfy many players (e.g., fans of the standard weapons in *Half-Life 2* may be disappointed to lose their entire arsenal when the "gravity gun" gets super-charged).

When the time comes to decide how a game's dynamics should change, how are such decisions made? For instance, why should the ghosts in *Pac-Man* slow down by only 50% instead of 75%? We investigate such questions in the context of Artificial Intelligence (AI), with the goal of taking *advantage* of different players' preferences to improve each player's experience. By learning from the additional data that becomes available when players start interacting with the game, an online adaptation agent can serve as an extra-informed proxy for the game's designers, carrying out their decisions in light of what it learns about each player (e.g., the AI Director used in *Left 4 Dead*). We focus on such agents for the remainder of this work, under the assumption that designers can make better decisions when they know more about their players on an individual basis.

In this paper, we present the Procedural Game Adaptation (PGA) framework: a designer-controlled way to adapt the dynamics of a given video game *during* end-user play. When implemented, this framework produces a deterministic, online adaptation agent (called an *experience manager* (Riedl et al. 2011)) that automatically performs two tasks: 1) it gathers information about a game's current player, 2) it uses that information to estimate which of several different changes to the game's dynamics will maximize some player-specific value (e.g., fun, sense of influence, etc.).

An Illustrative Example

As a high-level example of how procedural game adaptation (PGA) might proceed, we consider how it could change the dynamics of *Annara's Tale*, an interactive adaptation of the story of Little Red Riding Hood (Grimm and Grimm 1812). In an interactive story, the player is often cast as the protagonist in an ongoing (and usually fictional) narrative, where she repeatedly performs actions in response to the state of an interactive environment (i.e., a video game world). Whenever the player performs an action, the next state of the game is determined as a function of both its current state and the player's action. This function defines the dynamics of the game, and we refer to it as the game's *transition function*.

In *Annara's Tale*, a girl (called "Red" in Figure 1) meets a wolf in the forest on the way to her grandmother's house. At the top the figure, the player (as Red) was given the option

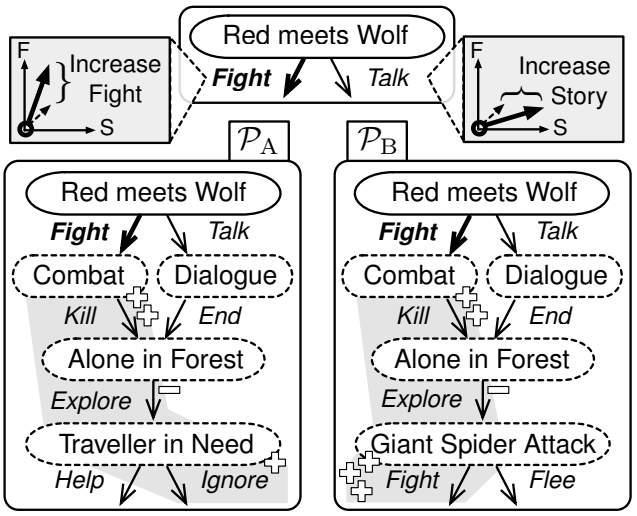

Figure 1: Comparing transition functions ($\mathcal{P}_A$ or $\mathcal{P}_B$) in *Annara's Tale* (Thue et al. 2010). Ovals are game states (dashed outlines show future states), and arrows are potential player actions (thick arrows show actions already taken). Shaded rectangles demonstrate learning player preferences toward different kinds of gameplay. Shaded paths show estimates of player actions, and $+/-$ symbols are estimates of the player's rewards for state/action pairs.

to either fight the wolf or engage it in conversation (actions *"Fight"* or *"Talk"* in the figure), and she decided to start a fight. Consider the point in time immediately after this action, with reference to both Figure 1 and the pseudocode in Figure 2. Whenever the player performs an action, the manager observes it along with the state in which it occurred, toward learning some player-specific information (lines 2 & 3 in the code). For example, choosing to fight the wolf instead of talking to it could indicate that the current player is more interested in fighting ("Increase Fight" in Figure 1).

After observing and learning about the player, the manager receives an opportunity to adapt the game's dynamics by modifying its transition function. Comparing the two transition functions that are available in our example ($\mathcal{P}_A$ and $\mathcal{P}_B$ in Figure 1), the only difference between them is the effect of the state/action pair $\langle$"Alone in Forest", *"Explore"*$\rangle$. Under $\mathcal{P}_A$, the player's action to explore leads from the state "Alone in Forest" to "Traveller in Need", but under $\mathcal{P}_B$, "Giant Spider Attack" will occur instead. Note that the manager chooses between $\mathcal{P}_A$ and $\mathcal{P}_B$ immediately after the player acts to fight the wolf, but the effects of its decision only become apparent once the player starts to explore the forest. In general, every change to the transition function can have both immediate and far-reaching consequences, potentially altering the effects of player actions taken much later in the game. In *Half-Life 2*, a PGA manager could learn the player's preferences for using the "gravity gun" versus more conventional weapons in Ravenholm, and then use that knowledge to change the results of her entering the Citadel later on: players who enjoy the gravity gun could be stripped of their conventional weapons as

```
1  for each player action
2      observe the action and its context state
3      infer player-specific information
4      estimate the player's rewards
5      estimate what the player will do
6      choose a new transition function
         based on the estimates
7  end for
```

Figure 2: Informal pseudocode for a PGA manager.

the gravity gun gets super-charged ($\mathcal{P}_A$), while players who preferred conventional weapons could receive powerful upgrades to them instead ($\mathcal{P}_B$).

Toward choosing a new transition function in a way that improves the player's experience, the manager first attempts to estimate the rewards that the player might derive from different potential future states and actions (line 4, $+/-$ signs in the figure), along with which actions she might take from the states that would occur with each transition function (line 5, and the shaded paths in Figure 1), In the example, given a prior player interest in combat (the steep upward vector resulting from "Increase Fight"), the manager might estimate that she would ignore a traveller in need and fight a group of giant spiders, and that she would find the latter to be the more rewarding of the two. Based on those estimates, it could choose to set the transition function to $\mathcal{P}_B$ (line 6), leading the spider attack to occur later on in the game.

Problem Formulation

In this section, we formally define the task of changing a game's dynamics in a player-specific way. We represent the player's experience in a video game as a Markov Decision Process (MDP), where the *player* is the agent that is situated therein. The manager operates *outside* the MDP, monitoring the player and modifying the MDP on the fly. The player knows her state in the MDP at all times, but she might not know the MDP's dynamics in advance; for the sake of clarity, we present the deterministic case only. Formally, the player's experience is a tuple, $\langle S, A, \mathcal{P}, \mathcal{R}, \pi \rangle$, where:

- S is a finite set of *states* ($s \in S$), each describing a different configuration of the game world and the player
- A is a finite set of possible player *actions* ($a \in A$), which cause the game to change from one state to another.
- $\mathcal{P} : S \times A \rightarrow S$ is a deterministic *transition function*, which, given a state $s \in S$ and action $a \in A$, gives the state s' that will occur at time $t+1$ if the player performs a in s at time t. The player can observe $\mathcal{P}$ for states and actions that occur, but may not know it otherwise.
- $\mathcal{R} : S \times A \rightarrow \mathbb{R}$ is the player's *reward function*, which, given a state $s \in S$ and action $a \in A$, gives the expected value (r) of the reward that the player receives at time $t+1$ after performing a in s at time t. The manager has no access to $\mathcal{R}$, and the player can only sample it at the states that occur during her experience.
- $\pi : S \rightarrow A$ is the player's deterministic *policy*, which, given a state $s \in S$, gives the action $a \in A$ that the player

will perform in response to s occurring. The manager can only observe π for states that happen during the game.

At each *time step* (t), the player perceives the current state, s_t. When she selects and performs an action according to her policy ($a_t = \pi(s_t)$), a new state occurs in the environment according to the transition function ($s_{t+1} = \mathcal{P}(s_t, a_t)$) as the time step advances (increasing by one from zero). The player then receives both the new state (s_{t+1}) and a reward as determined by the reward function ($r_{t+1} = \mathcal{R}(s_t, a_t)$).

We assume that the player takes actions in the environment (and thereby transitions between states in the MDP) to maximize her *return* – the sum of the rewards that she receives during the course of her experience ($\mathbf{r} = r_1 + r_2 + \cdots + r_n$), which we assume has a finite horizon. Brown et al.'s work in neuroscience (1999) offers support for treating players like reinforcement learning agents. From the designer's perspective, the return represents some particular aspect of the player's appraisal of the game (e.g., how much fun the player had, how challenged she felt, or how influential she felt her actions were); this aspect is what the manager aims to maximize during each player's experience.

This work is particularly targeted toward video games that have two characteristics: 1) that for a given player, different gameplay leads to different returns; and 2) that from a given set of gameplay, different players derive different returns. The presence of these characteristics respectively imply that 1) the player's return can indeed be maximized by controlling the dynamics of the game, and 2) to be successful, this maximization should occur differently for different players. These implications motivate our intent for the PGA framework: to enable managers to a) estimate the player's reward function and policy by learning from observations, and b) modify the transition function to maximize an estimate of the player's return. Before formalizing the manager, we present the constraints that limit its operation.

Designer Constraints To afford designers some control over the manager's operation, we allow them to constrain its choices between different transition functions. Formally, we represent all designer constraints using a single function, $\beta(s, a)$, called the *designer constraint function*:

- B is a set of *manager choices*, consisting of all possible transition functions for the player's MDP; and
- $\beta : S \times A \rightarrow 2^B$ is the *designer constraint function*, which, given a state and action in the player's MDP ($s \in S$ and $a \in A$), gives a subset of transition functions for the manager to make its next choice from.

Goal Given the foregoing constraints, the goal of procedural game adaptation is as follows: to maximize the player's return by repeatedly modifying the transition function of her MDP, subject to a set of designer constraints. Mathematically, the manager aims to compute:

$$\mathcal{P}_{t+1} = \underset{\mathcal{P} \in \beta(s_t, a_t)}{\arg\max}\ \mathbf{r} \qquad (1)$$

Related Work

This work has strong ties to two areas of game AI research: interactive storytelling, and dynamic difficulty adjustment.

The notion of experience management (also known as "drama management") has been explored extensively in the context of interactive storytelling, with AI managers taking actions to improve a player's experience in an interactive environment. The primary difference between PGA managers and others is the nature of the actions that they can perform; some managers directly initiate subsequent game states (Weyhrauch 1997; Roberts et al. 2006), while others issue directives to a set of semi-autonomous non-player characters (NPCs) (Riedl et al. 2011). By modifying the transition function, PGA managers subsume the direct selection of game states, and additionally offer designers the convenience of changing potentially large sections of gameplay with a single manager action. Similarly, directives can be "issued" to semi-autonomous NPCs by changing the part of the transition function that governs their behaviour.

The line of research following from Search-Based Drama Management (Weyhrauch 1997) defines a system of rewards that encode designer notions of experience quality, and later work also focuses the manager's efforts on a particular class of MDP (a TTD-MDP (Roberts et al. 2006)). Unlike PGA, however, these projects treat their *managers* as the agents that collect rewards and traverse the MDP, and use thousands of simulated runs to improve their ability to produce stories that follow a particular designer aesthetic. Conversely, PGA treats the *player* as the agent that interacts in the MDP and collects rewards for doing so, and it only uses the experience of its current (real) player to inform its manager's decisions.

Outside of interactive storytelling, interest in using AI to modify game dynamics online has increased over the past ten years, with the majority of efforts focusing on automatically adapting the game's difficulty to suit different players' skills (see (Lopes and Bidarra 2011) for a recent survey of this and other game-based adaptation). Such systems can be viewed as implementations of PGA, for they try to maximize a player rating that peaks when the game is neither too difficult nor too easy, and they do so by choosing between transition functions that vary the value or frequency of power-ups, or the number or strength of enemies. Andrade et al. (2006) apply a reward-centered learning algorithm to achieve similar results, but their approach differs from PGA in the same way as Search-Based Drama Management.

The PGA Framework

We now present the PGA framework in three parts: a learned player model, an estimate of the player's return, and an algorithm that combines the first two to perform PGA.

Learning a Player Model

Lacking any direct access to either the player's reward function ($\mathcal{R}$) or her policy (π), any attempt to maximize the player's return requires estimation. Given that $\mathcal{R}$ and π can both vary between players, every PGA manager estimates them by learning about each player individually: it observes her actions and their contexts in the game, and updates a collection of player-specific information that informs its estimations. We refer to this collection as the *player model*.

The purpose of the player model is to help the manager estimate $\mathcal{R}$ and π for states and actions that have not yet

occurred. To help generalize from the player's *previous* experience of states and actions as inputs, we represent the player model abstractly as a real-valued vector ($\vec{\mu}$) in an m-dimensional space. Each dimension represents the degree to which a particular statement (or its opposite, if negative values are used) applies to the current player (e.g., "the player is feeling tense"). The values in the model can be used to represent a wide variety of player-specific information, including estimates of her emotional reactions (e.g., stress, excitement), statistics that capture her level of skill (e.g., shooting accuracy), or observed inclinations toward or away from different styles of play (e.g., combat, puzzle-solving, etc.).

As a simple example, consider the model vectors shown in Figure 1. For the left model (the shaded box showing "Increase Fight"), the player has demonstrated a stronger inclination toward fighting than story-focused play (vector: $(5, 15)$). For the right model (showing "Increase Story"), the opposite is true. The number of dimensions used for the model (m) is a parameter of the framework; using too few may inhibit the manager's ability to distinguish between the different states and actions that it receives as its inputs, and using too many may hamper its ability to generalize its observations to novel states and actions.

The manager could have several opportunities to adapt the game's dynamics during the course of the player's experience. To ensure that each adaptation will benefit from the most informed estimates of $\mathcal{R}$ and π that are available at the time, it updates the model after every player action. It does so by calling its *model update function*, which retrieves specific expert knowledge about the game (namely: "How should each state/action pair in the game influence each of the m values in the model?") and uses it to update the current model. The initial model, $\vec{\mu}_0$, could be used to encode prior, player-specific information, or it could be set generally for all new players. Formally:

- $\mathcal{M} : M \times S \times A \rightarrow M$ is the *model update function*, where M is the set of possible player models. $\mathcal{M}$ can be defined by designers or data-mined from user experiences.

Given a player model, the manager uses it along with the structure of the game's MDP (i.e., $\langle S, A, \mathcal{P} \rangle \in D$, where D is the set of possible MDPs with $\mathcal{R}$ and π removed) to estimate the player's reward function and policy. It does so using two functions:

- $\mathcal{E}_{\mathcal{R}} : M \times D \rightarrow R$ is the *reward function estimator*, which estimates the player's reward function ($\tilde{\mathcal{R}} \in R$) given a player model ($\vec{\mu} \in M$) and an MDP with structure $\langle S, A, \mathcal{P} \rangle \in D$ (R is the set of all reward functions).
- $\mathcal{E}_{\pi} : M \times D \rightarrow \Pi$ is the *player policy estimator*, which estimates the player's policy ($\tilde{\pi} \in \Pi$) given a player model ($\vec{\mu} \in M$) and an MDP with structure $\langle S, A, \mathcal{P} \rangle \in D$ (Π is the set of all player policies).

Although one could assume that the player is rational and derive her policy from $\tilde{\mathcal{R}}$, we present $\mathcal{E}_{\pi}$ as-is to retain the framework's generality. We later provide example implementations of both $\mathcal{E}_{\mathcal{R}}$ and $\mathcal{E}_{\pi}$ in the context of two existing managers, and demonstrate how their outputs are used.

Estimating and Maximizing the Player's Return

The manager aims to optimize the player's experience by maximizing her return. Given current estimates of the player's reward function and policy ($\tilde{\mathcal{R}}_k$ and $\tilde{\pi}_k$) along with a state and action to start from ($s_k \in S$ and $a_k \in A$), the player's return can be estimated ($\tilde{\mathbf{r}} \approx \mathbf{r}$) with respect to a given transition function ($\mathcal{P}$) using Equations 2 and 3. The first sum in Equation 2 estimates how much reward the player has collected from the states and actions that have actually occurred, while the second sum estimates how much she might collect from now until the end of her experience.

$$\tilde{\mathbf{r}} = \sum_{t=0}^{k} \tilde{\mathcal{R}}_k(s_t, a_t) + \sum_{t=k}^{n-1} \tilde{\mathcal{R}}_t(s_{t+1}, \tilde{\pi}_t(s_{t+1})) \quad (2)$$

where future states and actions are generated according to:

$$s_{t+1} = \begin{cases} \mathcal{P}(s_t, a_t) & , t = k \\ \mathcal{P}(s_t, \tilde{\pi}_t(s_t)) & , t > k. \end{cases} \quad (3)$$

Toward maximizing the player's return, the manager compares candidate transition functions in terms of their effect on the estimated return. However, given that the player has already collected all of the rewards represented by the first sum in Equation 2, their values cannot depend on the manager's present choice between potential new transition functions. We can therefore omit that sum when writing the manager's maximization objective for optimizing the player's experience, as shown in Equation 4.

$$\underset{\mathcal{P} \in \beta(s_k, a_k)}{\arg\max}\ \tilde{\mathbf{r}} = \underset{\mathcal{P} \in \beta(s_k, a_k)}{\arg\max} \sum_{t=k}^{n-1} \tilde{\mathcal{R}}_t(s_{t+1}, \tilde{\pi}_t(s_{t+1})) \quad (4)$$

Although Equation 4 is complete in the sense that it includes an estimate of the player's reward for every state from s_{k+1} until the end of her experience, its accuracy relies on a sequence of estimates that may worsen as they stretch into the future. While this problem is shared by many kinds of lookahead search, in our case it is compounded by two additional facts: 1) the manager can never repair its estimates by observing the true reward function, $\mathcal{R}$, and 2) its estimates of both $\mathcal{R}$ and π will often vary from one time step to the next. These factors diminish the manager's likelihood of correctly predicting future states and actions, and so motivate a simpler approximation (where $k + \mathrm{L} \leq n$):

$$\underset{\mathcal{P} \in \beta(s_k, a_k)}{\arg\max}\ \tilde{\mathbf{r}} = \underset{\mathcal{P} \in \beta(s_k, a_k)}{\arg\max} \sum_{t=k}^{k+\mathrm{L}-1} \tilde{\mathcal{R}}_t(s_{t+1}, \tilde{\pi}_t(s_{t+1})) \quad (5)$$

In this approximation, we have preserved only the first L terms of the sum in Equation 4. If the reward function for the current player is sparse or generally homogenous, a discounted version of Equation 4 (putting less weight on more future-distant estimates) may be a useful alternative. Allowing non-determinism in both $\mathcal{P}$ and π would make the foregoing mathematics more complex, but our approach would remain unchanged. To simplify our presentation, we demonstrate Equation 5 with $\mathrm{L} = 1$ in Figure 5 below.

The PGA Algorithm

Input: $S, A, \mathcal{P}_0, s_0, \vec{\mu}_0, \mathcal{M}, \beta, \mathcal{E}_\mathcal{R}, \mathcal{E}_\pi$

1 **for each** time step $t \in [0, n]$
2 $s_t, a_t \leftarrow$ observe player state and action
3 $\vec{\mu}_{t+1} \leftarrow \mathcal{M}(\vec{\mu}_t, a_t, s_t)$
4 $\tilde{\mathcal{R}}_t \leftarrow \mathcal{E}_\mathcal{R}(\vec{\mu}_{t+1}, \langle S, A, \mathcal{P}_t \rangle)$
5 $\tilde{\pi}_t \leftarrow \mathcal{E}_\pi(\vec{\mu}_{t+1}, \langle S, A, \mathcal{P}_t \rangle)$
6 $\mathcal{P}_{t+1} \leftarrow \underset{\mathcal{P} \in \beta(s_t, a_t)}{\arg\max}\, \tilde{\mathcal{R}}_t(\mathcal{P}(s_t, a_t), \tilde{\pi}_t(\mathcal{P}(s_t, a_t)))$
7 wait for next time step
8 **end for**

Figure 3: Formal pseudocode for the PGA algorithm, which allows a manager to perform procedural game adaptation.

The PGA Algorithm

The manager changes the transition function immediately after observing the player's current state and action, but *before* the next state is generated. Figure 3 shows how learning the player model and estimating the player's return can be combined algorithmically, creating a manager that changes a game's dynamics in a player-informed way. The PGA algorithm governs the operation of every PGA manager.

In line 2 of the algorithm, the manager observes the player's action in the game (a_t), along with the state in which she performed it (s_t). In line 3, it uses the observed state and action with the model update function ($\mathcal{M}$) to update the current player model. In lines 4 and 5, it uses the reward function estimator ($\mathcal{E}_\mathcal{R}$) and player policy estimator ($\mathcal{E}_\pi$) to estimate a new reward function and player policy. In line 6, it retrieves a set of candidate transition functions using the designer constraint function (β), selects the one that maximizes its objective function (from Equation 5 with L = 1, while substituting $s_{t+1} = \mathcal{P}(s_t, a_t)$ from Equation 3), and applies it to the player's MDP. In line 7, the manager waits while player's MDP advances to the next time step, using $\mathcal{P}_{t+1}$ to generate the next player state.

An advantage of the PGA algorithm is that it can be seamlessly disabled and re-enabled. If for any reason the game's designers decide that the manager should *not* perform PGA for a certain portion of the game, it can be simply be switched off for a while and reactivated later; the game will continue operating as a normal MDP.

Initial Evaluation

We devised the PGA framework as a generalization of PaSSAGE, our academic interactive storytelling system with prior success in improving player enjoyment (Thue et al. 2010). By demonstrating that PaSSAGE is an instance of the PGA framework, we offer proof that the framework can be used to produce managers that achieve designer goals. Evaluating its representational efficacy and accessibility to designers remains as future work. In addition to examining PaSSAGE, we simultaneously demonstrate the PGA framework's versatility by presenting the AI Director from the critically acclaimed *Left 4 Dead* (Booth 2009) as an instance thereof as well. Both managers operate in the context of immersive 3D environments, with PaSSAGE's prototypes playing as story-focused role-playing games, and *Left 4 Dead* playing as a survival-themed cooperative shooter.

Action: a_t	State: s_t	Increment: $\vec{i}(a_t, s_t)$
Fight	Red meets Wolf	$(40, 0, 0, 0, 0)$
Talk	Red meets Wolf	$(0, 0, 40, 0, 0)$

Table 1: Part of PaSSAGE's model update function.

State: s_t	Increment: $i_x(s_t)$
Damage Taken by P_x	$C_1 \times$ *damage*
P_x Incapacitated	C_2
P_x Pulled off Ledge	C_3
Enemy Death near P_x	C_4/*distance*
No Enemies near P_x	$-C_5$
Otherwise	0

Table 2: Part of the AI Director's model update function, shown here for individual players. Every C_i is a positive constant.

MDP & Initial State: Figure 1 shows part of the story told by one of PaSSAGE's prototypes (*Annara's Tale*), demonstrating how PaSSAGE handles game states and player actions.

The game's initial state and transition function are set using simple triggers and scripts, as is common among many video games. In *Left 4 Dead*, this function implements a method for populating game areas with enemies ($\mathcal{P}_{\text{Build}}$), which adds enemies to increase the intensity of the game.

Initial Model & Update Function: PaSSAGE's player model is defined as a vector of inclinations toward five styles of play: ⟨Fighter, Method Actor, Storyteller, Tactician, Power Gamer⟩. Starting with neutral values for all styles (e.g., $\vec{\mu}_0$ = (F=1, M=1, S=1, T=1, P=1)), the model gets updated through designer-specified increments in the update function: $\mathcal{M}(\vec{\mu}_t, a_t, s_t) = \vec{\mu}_t + \vec{i}(a_t, s_t)$ (Table 1).

The AI Director's player model estimates the "emotional intensity" of up to four simultaneous players (P_1 to P_4). Starting with an assumption that every player's initial intensity is zero (e.g., $\vec{\mu}_0$ = (I_1=0, I_2=0, I_3=0, I_4=0)), the model update function causes each player's intensity to increase or decay based on her interactions with the enemies in the game: $\mathcal{M}(\vec{\mu}_t, a_t, s_t) = \vec{\mu}_t + \vec{i}(s_t)$ (Table 2).

Designer Constraint Function: PaSSAGE's designer constraints are used to restrict the set of plot events that can occur next in its interactive stories. For example, when the state/action pair ⟨"Alone in Forest", *"Explore"*⟩ is passed to the designer constraint function (β) of *Annara's Tale*, the result is the set $\{\mathcal{P}_\text{A}, \mathcal{P}_\text{B}\}$, whose values control the occurrence of either "Traveller In Need" or "Giant Spider Attack" (Figure 1). The designer constraints in *Left 4 Dead* allow the AI Director to control the pacing of the game, with β always evaluating to a set of two transition functions: $\{\mathcal{P}_{\text{Build}}, \mathcal{P}_{\text{Relax}}\}$ (as defined in the next section).

$mode(s_t)$	Model Test	$\tilde{\mathcal{R}}(s_t, a_t)$
Build Up	$\max_{\text{value}}(\vec{\mu}_{t+1}) < C_6$	$+1$
Relax	$\max_{\text{value}}(\vec{\mu}_{t+1}) > C_7$	-1

Table 3: Estimating player's rewards in *Left 4 Dead* for states that add new enemies under the given conditions. Rewards for all other states and conditions are zero. $C_6 > C_7$.

Reward Function Estimator: PaSSAGE estimates the player's reward function by comparing its current player model to the play-style increments that the designer specified for the given state/action pair, using an inner product computation: $\tilde{\mathcal{R}}_t(s_t, a_t) = \vec{\mu}_{t+1} \cdot \vec{i}(a_t, s_t)$; the more similar the two are, the higher the estimated reward.

Although the AI Director never forms an explicit estimate of $\mathcal{R}$, it still defines one implicitly through the finite state machine that controls its operation. During "Build Up" mode, enemies repeatedly spawn until the highest player intensity value exceeds a certain threshold (C_6), and continue for 3-5 seconds thereafter. The mode then switches to "Relax", during which no new enemies spawn, but existing enemies may still attack. Once the highest player intensity value has decayed below another threshold (C_7), a 30-45 second delay occurs before the mode switches back to "Build Up" and the cycle begins again. Using the PGA framework, this pattern can be represented by the estimated reward function in Table 3, disabling the manager for the required delay duration when each intensity threshold is reached.

Player Policy Estimator: PaSSAGE's player policy estimator ($\mathcal{E}_\pi$) returns a policy that is tied directly to the estimated reward function, under the assumption that players will perform whichever action seems the most rewarding at the time: $\tilde{\pi}_t(s_t) = \arg\max_{a \in A} \tilde{\mathcal{R}}(s_t, a)$. Given that the AI Director could estimate the player's reward function using only her current state, it need not estimate her policy.

Discussion, Limitations, and Future Work

From a designer's perspective, the primary advantage of the PGA framework is that it represents a conceptual simplification of achieving player-specific adaptation via more traditional means. Specifically, while the effects of any PGA manager can be duplicated using only a basic MDP (e.g., by integrating the player model and its updates into the state and action sets), doing so would vastly increase the effective number of unique states and actions that the designer would need to consider. By factoring the task of player-informed adaptation into two components that are separate from the MDP (namely, learning a player model and estimating the player's return), the problem becomes easier to approach in a way that leverages AI, but retains designer control.

In addition to serving as a template for creating new, player-informed managers, the PGA framework offers a common foundation for the analysis and comparison of existing work. We have demonstrated this potential with our representation of PaSSAGE and the AI Director from *Left 4 Dead*, but it would be very interesting to see other PGA-related managers being investigated in a similar way.

Although we have intentionally presented the simplest and most general version of the PGA framework in this paper, the result of doing so is that there may be several opportunities to optimize its implementation. For example, it is not necessary to explicitly define designer constraints for every state/action pair in a game: by causing the manager to assume that $\beta(s_t, a_t) = \{\mathcal{P}_t\}$ unless it is told otherwise, designers can focus on the particular places where procedurally changing the game's dynamics will have the greatest effect. Furthermore, if $|\beta(s_t, a_t)|$ is usually small in comparison to $|S \times A|$, then efficiency can be improved by recomputing only the values of $\tilde{\mathcal{R}}$ and $\tilde{\pi}$ that correspond to the states and actions that are relevant to β. Estimating the player's reward function and policy is generally non-trivial, but Syed et al.'s work on Apprenticeship Learning (2008) has made promising advances in this regard.

Looking forward, PGA could potentially be used to create an AI assistant for the game design process. Given a set of player models (perhaps data-mined from a previous game), $\mathcal{E}_\pi$ could be used to create simulated players for a prototype game, and Equations 4 or 5 could be used to estimate returns for transition functions from a very loosely constrained set. By clustering the resulting data to find a handful of canonical models, the top transition functions for every model could be thought of as the best candidates for inclusion as designer constraints in the next iteration of development. Such a method might help developers manage the tradeoff that exists between reaching a wider audience and having to create more game content to do so, identifying the smallest set of transition functions that would be needed to reach a given range of players.

Conclusion

In this paper, we investigated the problem of automatically changing the dynamics of a video game during end-user play. Moving beyond traditional, *ad hoc* methods for experience manager design, we proposed a new framework, Procedural Game Adaptation, that is both theoretically principled (using the formalism of MDPs) and practically applicable in the contexts of academia (PaSSAGE) and industry (the AI Director). Treating the player's experience as an MDP, managers created using our framework automatically learn a model of each player and use it to estimate and maximize her return, all the while operating within a set of designer-specified constraints. By dynamically changing the transition function of the player's MDP, PGA managers have the ability to affect large portions of gameplay with a single operation, and can leverage player information that only becomes available once gameplay has begun.

References

Andrade, G.; Ramalho, G.; Gomes, A. S.; and Corruble, V. 2006. Dynamic game balancing: an evaluation of user satisfaction. In *AIIDE 2006*, 3–8. AAAI Press.

Booth, M. 2009. The AI systems of Left 4 Dead. *Keynote Presentation at AIIDE 2009*.

Brown, J.; Bullock, D.; and Grossberg, S. 1999. How the basal ganglia use parallel excitatory and inhibitory learning pathways to selectively respond to unexpected rewarding cues. *Journal of Neuroscience* 19:10502–10511.

Grimm, J., and Grimm, W. 1812. Little red cap. In *Kinder- und Hausmärchen, 1st ed*, volume 1.

Lopes, R., and Bidarra, R. 2011. Adaptivity challenges in games and simulations: A survey. *IEEE Transactions on Computational Intelligence and AI in Games* 3(2):85–99.

Riedl, M. O.; Thue, D.; and Bulitko, V. 2011. Game AI as storytelling. In Calero, P. G., ed., *Artificial Intelligence for Computer Games*. New York: Springer USA. 125–150.

Roberts, D. L.; Nelson, M. J.; Isbell, C. L.; Mateas, M.; and Littman, M. L. 2006. Targeting specific distributions of trajectories in MDPs. In *AAAI-06*, 1213–1218. AAAI Press.

Syed, U.; Schapire, R.; and Bowling, M. 2008. Apprenticeship learning using linear programming. In *Proc. of the 25 International Conf. on Machine Learning*, 1032–1039.

Thue, D.; Bulitko, V.; Spetch, M.; and Webb, M. 2010. Socially consistent characters in player-specific stories. In *AIIDE 2010*, 198–203. Palo Alto, California: AAAI Press.

Weyhrauch, P. 1997. *Guiding Interactive Drama*. Ph.D. Dissertation, Carnegie Mellon University, Pittsburgh, PA.