Player Modeling: Dynamic Difficulty and Playstyle Adaptation

Samuel Schimmel

1 Introduction

Player modeling is a form of indirect adaptation that involves recording player behavior and using the resulting data to customize the gameplay experience. This paper will discuss the implementation of a player modeling system in an Unreal Engine 4 first person shooter, player modeling as a data-driven user testing tool, modeling player performance to provide dynamic difficulty, and modeling player preferences in order to encourage players to try new content and strategies.

2 Implementation

The first step in player modeling is to write a PlayerModeling manager class to receive updates, store data, time intervals, calculate moving averages, and provide statistics to the rest of the program. PlayerModeling should be initialized as a singleton when the session begins, and persist between level loads.

Statistics

We store our data in a map, with statistic names as the keys and instances of our statistic struct as the values. The statistic struct has fields for all of the contexts in which we need to represent our data, including the current interval (a fixed-length window of data collection), the moving average, the player's current life, the current checkpoint, the current objective, the current level, and the whole campaign.

We support two kinds of statistics – static and dynamic. Static statistics are specified in code. Some of these are one-dimensional (e.g., kills, deaths, time, etc.), while others are composites of multiple statistics. For example, performance is a composite of damage dealt and damage taken. Accuracy is a composite of projectiles fired and projectile hits.

Dynamic stats, on the other hand, are generated at runtime, the first time our data tracking function is called with a static statistic and the name of a Blueprint class (i.e., a specific piece of game content). For example, we had to program our system to track `DamageDealt`, but having done so it will automatically track `DamageDealtWithPistol`, `DamageDealtWithShotgun`, and so on, without any additional code. This is beneficial in a content-heavy game where designers are adding, changing, and removing weapons all the time.

The Track Function

The most important public function in our player modeling class is the Track function. This is the interface that the rest of the program uses to notify player modeling about player actions. Our Track function takes the name of the statistic to be updated as a string,[1] a delta as a float, and an optional void pointer.[2]

When Track is called, it looks up the given statistic in the map and updates its interval value using the given delta (as well as the values for current life, current objective, etc.). At the end of each interval, each statistic's interval value is used to recalculate its moving average value, then reset.

The Moving Average Equation

The moving average equation is ideal for player modeling because it represents recent behavior – each new observation has the same weight regardless of sample size, and old observations don't continue to skew data long after the player's behavior has changed. For example, although we track the player's lifetime performance for the campaign as a whole, we don't use it for dynamic difficulty. If we did, taking an enormous amount of damage at the start of the game would continue to make the game easier thirty hours later. The moving average is given by the following equation: average = $\alpha \cdot$ newObservation + $(1 - \alpha) \cdot$ average. α, the learning rate, is usually 0.05 if changes are common, or 0.01 if changes are rare.

We initially considered using a moving average window instead of the moving average equation. The moving average window simply maintains a list of the last *n* observations. Using a moving average window would offer some advantages, including allowing us to calculate the median, allowing us to discard outliers, and allowing us to analyze change over time. However, we wanted to avoid the complexity and overhead of maintaining such a window and sorting it to calculate medians.

3 Data-driven User Testing

Before discussing how we can use statistics to dynamically adapt to players, it's worth mentioning the value inherent in the statistics themselves. We found it helpful to create a UMG widget (an Unreal Engine 4 UI object) for displaying all the values of a given statistic at runtime. A console command, which takes the name of the desired statistic as a parameter, toggles the widget and makes the given statistic the current debug statistic so the widget knows which data to request from the player modeling class. Ideally, one could view this debug output on a second monitor while administering a playtest. The statistics can also be

1 We originally had a statistics enumeration, but switched to a namespace of static const strings to facilitate adding new statistics at runtime.

2 The void pointer is for passing structs of more detailed data as needed. This will also allow us to expand the system in the future without changing the interface.

serialized to a file at the end of each session, uploaded to a server, and used as samples in meta-analysis of multiple playtest sessions.

4 Dynamic Difficulty

Dynamic difficulty is driven by the performance statistic. Performance is a composite of the damage dealt and damage taken statistics. Damage dealt measures damage dealt to enemies by the player. Damage taken measures damage taken by the player, from enemies (this makes it harder for players to damage themselves to exploit the system and make the game easier).

Which statistics you use to model player performance will depend on your game design. For example, we considered taking accuracy into account, but decided against it because our game is a fast-paced retro shooter with little emphasis on precision. For a slow-paced survival-horror game, using accuracy to model player performance would be more reasonable.

Calculating Performance

The performance moving average is calculated at the end of each interval. First we get the damage dealt and damage taken interval values. If they are both 0 (i.e., no combat occurred that interval) we return early. Otherwise, we multiply them by weights which are determined by the current difficulty setting (more on that soon).

After weighting damage dealt and damage taken, we find their sum, min, max, and range. The performance interval value is given by `(sum – min) / range`. Finally, we use the performance interval value to recalculate the performance moving average.

The Player Damage Multiplier

The performance moving average is used to calculate a value by which all damage to the player is multiplied. Our designers specify a minimum and maximum for these values. At the end of each interval, we set the multiplier to `minPlayerDamageMultiplier + (performanceMovingAverage * range)`. Range is difference of the minimum and maximum damage multiplier values.

In our game, the performance moving average only influences the player damage multiplier. We considered having it affect an enemy damage multiplier as well, but decided that players would be more likely to notice if we adjusted how many hits it took to kill enemies. In different games, the performance moving average could influence all kinds of variables, including item drops, enemy spawns, enemy reaction times, and enemy perception.

Difficulty Modes and Performance Weights

Like most first person shooters, our game supports traditional difficulty modes (easy,

normal, hard, etc.), but they don't directly make the game easier or harder. Instead, they determine the damage dealt and damage taken weights, which are used in calculating the performance moving average, which is used to calculate the player damage multiplier.

For the designer authoring these weights, a good heuristic is to ask, "for a hypothetical player of a given skill level, what is the expected ratio of damage they will deal to damage they will take in our game?" This can be a challenging question to answer, but the data-driven user testing capabilities of player modeling can help.

If an average player will deal about as much damage as they take, the damage dealt and damage taken weights for normal mode should be set to 1 and -1, respectively. Therefore, on normal mode, playing well and playing poorly will affect performance equally.

If a beginner will take about 10 times as much damage as they deal, the weights for easy mode should be 1 and -10, respectively. Therefore, on easy mode, playing well will slowly increase performance, but playing poorly will decrease it quickly.

If an expert will deal about 10 times as much damage as they take, the weights for hard mode should be 10 and -1, respectively. Therefore, on hard mode, playing well will increase performance quickly, and playing poorly will decrease it slowly.

In other words, difficulty modes influence the elasticity of the performance moving average. You can get the hard experience on easy mode, but only if you play extremely well. Likewise, you can get the easy experience on hard mode, but only if you play extremely poorly.

An Alternative Implementation

In retrospect, it would have been valid to make the performance weights static and let our difficulty modes control the minimum and maximum player damage multipliers (instead of the other way around, as we implemented it). This would make design data more intuitive to author, as designers would not have to come up with performance weights for each difficulty mode.

The choice of which implementation to use should be answered by the following question: who should have more control over the difficulty? Our implementation maximizes AI control, makes difficulty fully dynamic, and treats difficulty modes as inputs into its algorithm. The alternative – tying the player damage multiplier to difficulty modes – maximizes player control and allows for greater variance between difficulty modes. In other words, our implementation lets the AI move the difficulty all the way from easy mode to hard mode, whereas the alternative would only let the AI adjust the difficulty *within* easy mode or hard mode. Which option is best will depend on your game design.

5 Playstyle Adaptation

In addition to modeling player performance, we can also model player preferences. First person shooters (and immersive sims in particular) frequently offer players multiples ways to play – they can choose between lethal or non-lethal abilities, combat or stealth approaches, and using weapons or superhuman powers. Collecting data on which content and strategies players prefer, and which are most effective, is valuable in and of itself. However, we can also use the data to dynamically adapt to player preferences.

Our game features a variety of interesting weapons, and we want to make sure players are using them all, rather than picking a favorite and sticking with it exclusively. In our game, players mainly acquire new weapons by taking them from fallen enemies. Enemies spawned without weapons are assigned a weighted random weapon that the player has used before but has been using less recently. At the end of each interval, after updating moving averages of damage dealt for each weapon, we sum these moving averages, then divide each moving average by the sum and subtract the quotient from 1. This gives the probability of each weapon being the next weapon to spawn with a new enemy. Weapons the player has been using a lot recently have lower probabilities, and weapons the player hasn't been using as much recently have higher probabilities.

The net effect of this algorithm is to encourage players to experience all of the game's content. The algorithm is only used if level designers choose not to assign an enemy a weapon, so they retain the flexibility to customize specific encounters (e.g., giving an enemy in a tower a sniper rifle). Furthermore, the algorithm only considers weapons that the player has used before, so designers also retain control over when content is introduced.

6 Conclusion

The use cases for player modeling aren't limited to data-driven user testing, dynamic difficulty, and playstyle adaptation. It can also be used for player feedback, either in real-time (i.e., dynamic narration), after levels (i.e., characterizing players' playstyles and comparing them to other players), or at the end of the game (i.e., using player modeling to choose between multiple endings).

Player modeling can also be used in non-combat games. A first person exploration game could model when and for how long a player looks at objects of interest simply by raycasting from the camera to the object. Similarly, it could model when and for how long the player looks at interesting views by tracking their time inside a collider with a specified direction vector. Adventure games can use player modeling to determine if a player is stuck on a puzzle, and offer hints accordingly. Its myriad applications in games of all genres make player modeling a rich and versatile technique.