Adaptive Mario : CS8803 Game AI Project 3

Woody Folsom and Marshall Gillson

March 18, 2012

1 Introduction

The game mechanics of the Super Mario world are celebrated and well-understood. Indeed, even in this age of super-computer graphics and Hollywood-level production values, few games can match the pure enjoyment delivered by this simple 8-bit Nintendo platform jumper. Here, we endeavor not to change that formula, but to imbue it with a modern, artificial-intelligence driven approach to enhance the game-play experience.

In creating Adaptive Mario, we use several procedural content generation (PCG) techniques built upon this simple platform-jumper engine. Our algorithms adjust to player preferences and skills, and guide players toward an ideal game-play experience. Our design is intended to ensure that the maximum amount of replay value is delivered, considering the limited multimedia assets and game-play mechanics of the Infinite Mario engine on which it is based.

Every Mario level has the implicit goal of progressing from left to right until the completion gate appears, all the while keeping Mario safe. A number of more esoteric subgoals also arise, such as destroying enemies, collecting coins, and minimizing time of completion. Our conceptual design goal, however, was a bit different. As shown in Figure **??** we aim to provide players with different styles of play and skill levels with distinct and customized game-play experience from the moment the level begins. The process of generating a level consists of a number of steps. First, a player profile is drawn based on the players performance in an average, non-customized level. That data is fed into our generation engine to guide the creation of subsequent levels. The level generation pipe-line is explained in full detail in *Section 3 Approach*.

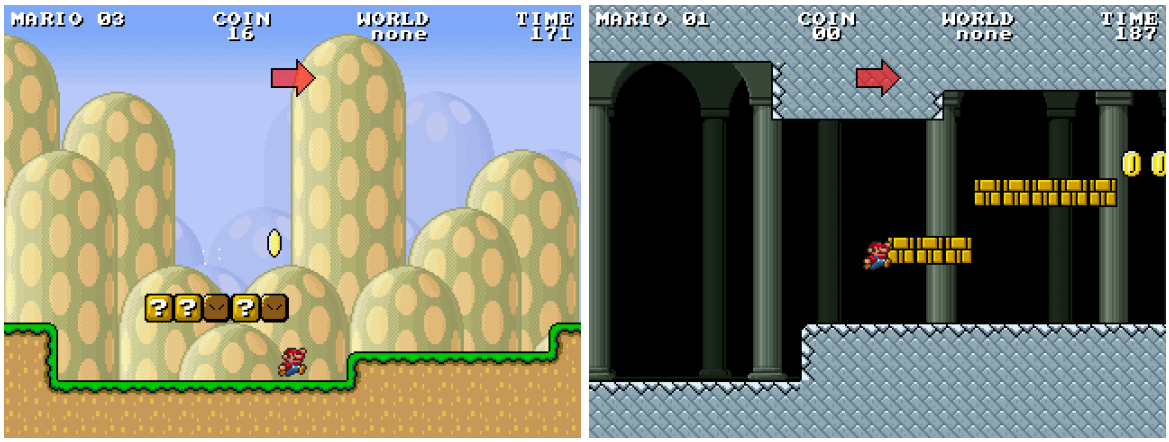

Figure 1: Adaptive Mario in action

As shown in Figure 1, the conceptual goal is that players with different play styles and skill levels should have a very different game-play experience from the moment the level begins. The very first time Adaptive Mario is run in a new environment, a random level is presented based on an a priori model of an 'average' player. A high degree of randomness at this point allows the player to freely choose a game-play style rather than being constrained by a prefabricated environment.

2 Related Works

A fundamental concept in the design of Adaptive Mario is *flow*. As defined by Mihly Cskszentmihlyi, according to Salen and Zimmerman [1], flow is a state of total immersion and concentration in which the player believes he or she is overcoming obstacles by the narrowest margins. As shown in Figure 2, achieving this state involves a delicate balance between the difficulty of the game and the degree of the player's skill. If the game is too hard, the player will become frustrated. On the other hand, most players will become bored if the game is too easy. Our overarching goal was to facilitate a flow state by giving the player sufficient challenge and variety without rapidly becoming too difficult.

This theory presents two variables: challenge and skill. Adapting the difficulty of a level is well within our purview as game designers, but the level of difficulty required to generate flow state depends on the players skill level. We therefore

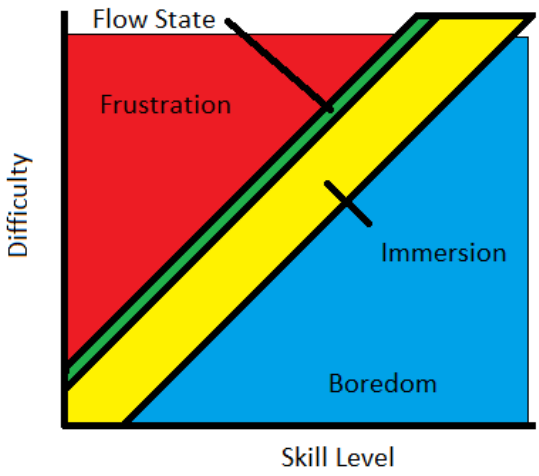

Figure 2: Flow State: Game Difficulty vs. Player Skill

first developed a model of player psychology and ability. Numerous such models have been proposed since Richard Bartle's 1996 study of MUD (Multi-User Dungeon) players [2], all of which attempt to classify a player into a category or type. In developing our model, we considered some notable examples:

Author	Player Model
Richard Bartle	Killer, Socializer, Achiever, Explorer
Nick Yee	Relationships, Immersion, Grief, Achievement, Leadership
John Radoff	Immersion, Cooperation, Achievement, Competition

The Adaptive Mario PlayerProfileMatcher class assigns the player one of five roles based on past performance: BUMPER, COLLECTOR, JUMPER, RUNNER or SHOOTER. This model is loosely based on Bartle's extended model, with BUMPER corresponding to Planner, in the sense that the player is observed to interact with the environment by bumping bricks and throwing shells. However, our model necessarily differs somewhat from the above examples, as Adaptive Mario is a single-player game with no social element.

3 Approach

The Adaptive Mario level generator functions as a 4-stage pipeline composed of a Profile Matcher, Level Archetype Selector, Level Generator and Challenge Component Creator.

Profile Matcher

The first step of the level generation process is to evaluate the player's style and skill level. Adaptive Mario uses the output file "player.txt" generated by the GamePlay class during the most recent game, if available. If no player data is available, as in a first run of the game, a simple, un-weighted level is generated, to assist in collecting baseline player data for future runs. More detailed statistics from the DataRecorder class are preserved as XML and are used by some components of the level generator. The ProfileMatcher scores aspects of game-play such as number of kills, jumps, and coins collected in order to evaluate the player's skill in the five identified areas of competency and generate a vector of skill values, within a range of $1 - 100$. The relative weight of each different skill is maintained by the player profile for reference by later segments of the level creation process. Here, the player is also assigned a type based on his or her most dominant trait, and an overall skill level: expert, proficient, competent, beginner, or novice. A players skill level is based on a number of different factors. First of all, his or her average skill level across the skill vector is included. The speed with which the previous level was completed is integrated, as a player who finds a level too easy can complete it quickly and without error. Finally, the number of times their power-up state changed; if a player is not being threatened or hurt will tend to remain in one power-up state through much of a level, with few changes. A weighted overall skill score is produced and mapped to one of the five skill levels as shown below.

These categories are based on the Dreyfus model of skill acquisition with each successive level representing an achievement of 20% of the Profile's key skill maximum score, before penalties. Actions such as dying or making unnecessary jumps negatively impact the skill assessment.

Score	Skill Level	Attributes
20%	Novice	Low situational awareness, Reflexive responses
40%	Beginner	Uses judgment to react to challenges, Limited awareness
60%	Competent	Copes with multiple challenges, Uses sound strategy
80%	Proficient	Makes rapid decisions, Prioritizes goals
100%	Expert	Intuitively solves challenges, Pushes limits

Level Archetype Selection

The Level Archetype selector picks an overall macro-form for the level based on the player's profile and skill level. As there are five categories and only three distinct environments in the Inifinite Mario engine (Overland, Underground and Castle), different profiles cause similar Level Archetypes to be generated. However, this does provide the opportunity to indicate to the player what types of challenges lie ahead and may serve to enhance immersion.

Next, the Level Archetype is further customized for the player by disabling challenges and enabling rewards according to a predefined set of rules. In order to allow the rule set to be easily customized and extended, Adaptive Mario uses Drools, an implementation of Charles Forgy's 1982 Rete algorithm [3]. Once the appropriate rules have fired, level generation proceeds to the third stage: macro-structure generation.

Level Generation: Macro-structure

The task of the macro-structure generator is to deliver a list of sequential Mario level design elements (LevelComponents) based on a high level description similar to "Overland level with gaps but no pipes, difficulty 5/10". Several algorithms were considered during the planning phase of this module. A Genetic Algorithm approach could potentially produce novel levels, but judging the relative fitness of candidate levels could be problematic. Hierarchical task networks (HTNs) and rhythm-based approaches have also been used, but tend to generate levels with a very uniform structure in a single pass. For ease of modification and to leverage the authors' design intuition, a context-free grammar (CFG) was implemented instead.

Given a suitable array of predefined LevelComponent types, each of which corresponds to one or more Infinite Mario game sprites, the structure of a level archetype can be completely specified in the form of a file-based CFG. As shown in the following example, the OR clauses are assigned probability values, giving a stochastic property to the generated level structure.

overland.grm

```
1 #VAR name = LevelComponent.TYPE
2 VAR S = LEVEL
3 VAR LAND_SEGMENT = LEVEL_SEGMENT
4 VAR LO_HI = LO_HI
5 VAR HI_LO = HI_LO
6 VAR LO_PATH = LO_PATH
7 VAR HI_PATH = HI_PATH
8 VAR lo_path = FLAT_LO
9 VAR hi_path = FLAT_HI
10 VAR pipes = PIPE_JUMP
11
12 #RULE name -> {probabilities}, (clause) [+,|] (clause)...
13 RULE S -> LAND_SEGMENT + LAND_SEGMENT
14 RULE LAND_SEGMENT -> {0.25,0.65,0.10}, (LO_HI + HI_LO) | (LO_PATH) | (LAND_SEGMENT + LAND_SEGMENT)
15 RULE LO_HI -> LO_PATH + HI_PATH
16 RULE HI_LO -> HI_PATH + LO_PATH
17 RULE HI_PATH -> {0.25,0.75}, (HI_PATH + HI_PATH) | (hi_path)
18 RULE LO_PATH -> {0.10,0.60,0.30}, (LO_PATH + LO_PATH) | (lo_path + pipes + lo_path) | (lo_path)
19
20 #START variable name
21 START = S
```

Figure 3: A Simple Overland Level Grammar

One potential pitfall of using a CFG for this purpose is that the generated level may be over-specified (containing too many elements) and hence too crowded, or under-specified and nearly empty. To mitigate this problem, a fitness evaluation function iteratively invokes the LevelGrammar class's generateRandomTree() method, rejecting proposed levels with too many or too few LevelComponents.

Challenge Components: Micro-structure

Challenge components are the building blocks of an Adaptive Mario level. Each challenge component represents a small puzzle or level element. Each level consists of a sequence of challenge components. Through only a small number of these

components, we have achieved a wide variety of possible levels. In fact, the challenge components induce variety and adapt to the players style in the following ways.

First, the selection of components plays a large role in defining the character of a level. Many challenges are better suited for different play styles, based on the skills they require to traverse or the rewards they cause a player to incur. This allows a level to shift toward a more engaging landscape for a player of a particular style. Second, each challenge component dynamically adjusts its difficulty based on the players skill level. The players skill competencies affect the number of enemies they encounter, the number of power-ups they are offered, the number of coins placed on a level, the difficulty of the enemies they face, the size and number of pits in a level, and the types of terrain encountered. Because challenge components are so dynamic, even a single challenge element encountered repeatedly can present and engaging challenge.

An additional benefit of challenge components is their modularity. Though we have implemented a strong suite of challenges, innumerable more are possible. With the implementation of a function and slight modification of the level generation grammar, new challenges can be added to Adaptive Mario to make it even more variable and extend its replay value further.

Difficulty	Challenge Component	Description
1	Coin Dive	Some empty blocks, coins line the path to the ground.
2	Free power-up	Sets the player up to get a power-up with little or no challenge.
3	Straight	A straight stretch of land with maybe one enemy and maybe some coins or blocks
4	Single Pit	A pit that the user must jump over; rocks on either side.
5	Bowling Alley	A red koopa right before a long line of enemies. Kill them all!
6	Cannon Line	A stack of 2 or 3 cannons.
7	Maze	A maze of indestructible blocks, with enemies.
8	Lemming Trap	A little pit with a few enemies that jump down into it.
9	Platform Jump	A bunch of platforms to jump between.
10	Pipe Jump	A bunch of thin pipes to jump between.

4 Evaluation

Developing Adaptive Mario presented several challenges. From a technical standpoint, finding a Java-compatible Rete rule engine became a time-consuming effort, due mostly to the large number of third-party JAR dependencies. Ultimately, we were able to incorporate the Drools engine into our code base. The amount of time, however, that was invested in doing so forced the minimization of role of the rules engine. It therefore makes only basic alterations to random levels based on the player profile.

Randomized level generation using a stochastic CFG was largely successful, however. Not only can our very simple grammar generate a huge variety of levels, the implementation of a parser for the rule set allows future designers to make meaningful content changes without needing to recompile source code. Such grammars are, of course, limited to some degree. The grammar by definition lacks any contextual awareness. It therefore does nothing to avoid awkward juxtaposition of challenge components. Though challenge components are self-contained such that transition between any two should not present a problem, we nonetheless recognize this stratification as a limitation of our architecture.

Perhaps the most challenging aspect of our development process was the creation of the PlayerProfile. Each level is generated based on player data from only the single previous level traversal. Thus, the terrain of a previous level may have an inordinate effect on the evaluation of a players current tastes. We built our ProfileMatcher to minimize this possibility, but it nonetheless exists. A more complex version of Adaptive Mario would benefit greatly from a more iterative and flexible approach to player modeling, perhaps by employing an artificial neural network or Bayesian inference. These techniques would allow for more accurate and therefore useful scoring values, but would require more involved modification of the game engine.

5 Conclusion

It seems clear from these results that the framework of Adaptive Mario has the potential to guide the player toward his or her own idealized version of a platform game, while still presenting a reasonable level of challenge. Not only does difficulty scale in proportion to the player's performance (preventing frustration), but care is taken in the design of the level grammar to avoid repeatedly giving the player 'more of the same' (leading to boredom). We certainly recognize many areas ripe for improvement (such as the limited level grammar), yet Adaptive Mario as currently implemented represents a robust engine. It provides tools for the adaptive content designer and new experiences for the consumer. It certainly adds a novel twist on the genre for any platform gaming enthusiast.

Appendix A: Building the Game

Building the Project 3 executable requires a Java SDK version 1.6+ and Apache Ant.

To build the game, execute `ant clean` followed by `ant` from the main project directory.

Appendix B: Running the Game

To run the game, change to the 'dist' subdirectory following a successful build and execute `java -jar CS8803_P3.jar`.

References

[1] K. Salen and E. Zimmerman. *Rules of Play.* The MIT Press, 2004.

[2] Richard Bartle. Hearts, clubs, diamonds, spades: Players who suit MUDs. http://aigamedev.com/open/interviews/mario-ai/, 1996. [Online; accessed 18-March-2012].

[3] C. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19:17–37, 1982.