Adaptive Mario : CS8803 Game AI Project 3

Woody Folsom and Marshall Gillson

March 18, 2012

Introduction

The authors of Adaptive Mario use several procedural content generation (PCG) techniques to develop a simple platform-jumper engine which adjusts to player preferences and skills to guide the player toward the ideal game-play experience. The Adaptive Mario design is intended to ensure that he maximum amount of replayability is delivered, considering the limited multimedia assets and game-play mechanics of the Infinite Mario engine on which it is based.

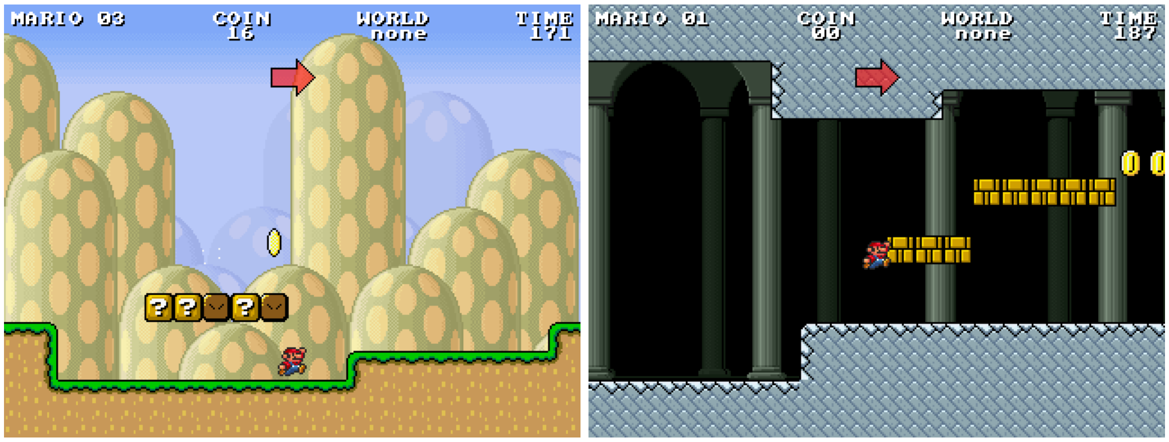


Figure 1: Adaptive Mario in action

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.

Related Works

The essential goal of Adaptive Mario is to facilitate flow state by giving the player sufficient challenge and variety without rapidly becoming too difficult. As defined by Mihly Cskszentmihlyi, according to Salen and Zimmerman [1], "flow" is the state of total immersion and concentration in which the player believes he or she is overcoming obstacles by the narrowest of margins. As shown in Figure 2, achieving this state involves a delicate balance between the difficulty of the game and the player's degree of 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.

In order to gauge the requisite level is difficulty, it is first necessary to develop a model of the player's ability. Numerous models of player psychology have been propossted since Richard Bartle's 1996 study of MUD (Multi-User Dungeon) players [2]. Some examples are:

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

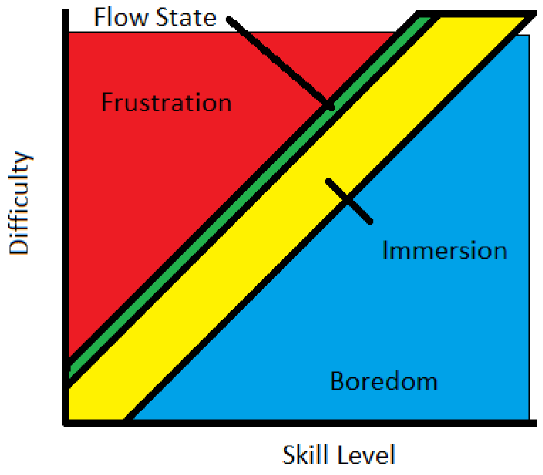


Figure 2: Flow State: Game Difficulty vs. Player Skill

throwing shells. This model differs from the examples above primary in that social elements are not evaluated, since Mario is a single-player game.

Level Generator Design

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. In addition, 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 areas of running, jumping, collecting, shooting. This vector of skills, rated on a range of 1 - 100, is then compared against a pre-calculated 'typical' vector for each of the five player types using a simple mean-squared error metric. The player is then given the profile matching the most similar set of skills.

In addition to assigning a player profile, the ProfileMatcher assesses the player's skill level by comparing specific metric against an ideal score assigning the player one of five skill levels: Novice, Beginner, Competent, Proficient or Expert. These categories are based on the Dreyfus model of skill acquisition with each successive level representing an achievement of 20

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

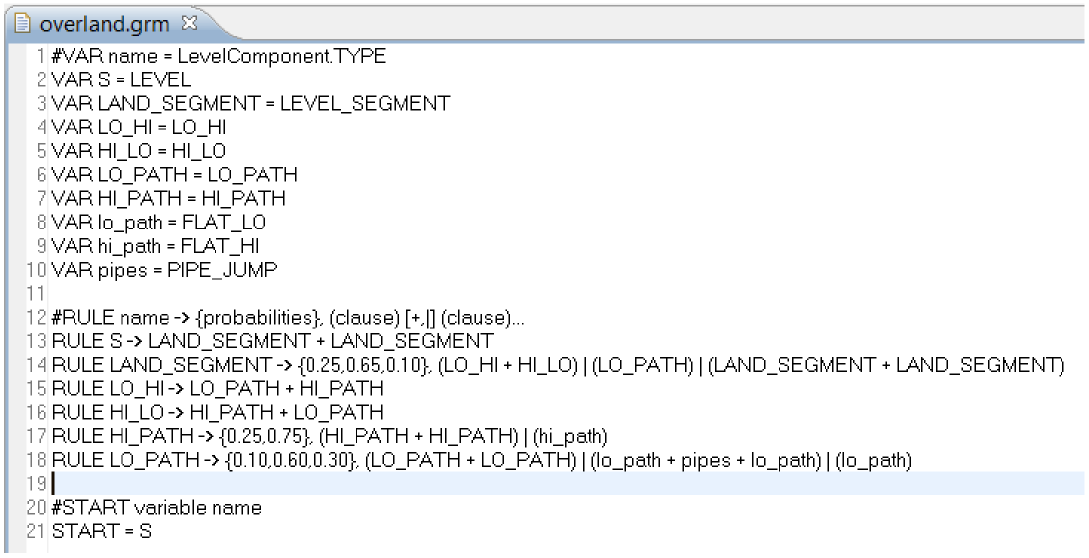
The Level Archetype selector picks an overall macro-form for the level based on the player's profile and skill level. As there are five categories and only three distinct environments in the Inifinite Mario engine (Overland, Underground and Castle), different profiles cause similar Level Archetypes to 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.



```
#VAR name = LevelComponent.TYPE
VAR S = LEVEL
VAR LAND_SEGMENT = LEVEL_SEGMENT
VAR LO_HI = LO_HI
VAR HI_LO = HI_LO
VAR LO_PATH = LO_PATH
VAR HI_PATH = HI_PATH
VAR lo_path = FLAT_LO
VAR hi_path = FLAT_HI
VAR pipes = PIPE_JUMP

#RULE name -> {probabilities}, (clause) [+,|] (clause)...
RULE S -> LAND_SEGMENT + LAND_SEGMENT
RULE LAND_SEGMENT -> {0.25,0.65,0.10}, (LO_HI + HI_LO) | (LO_PATH) | (LAND_SEGMENT + LAND_SEGMENT)
RULE LO_HI -> LO_PATH + HI_PATH
RULE HI_LO -> HI_PATH + LO_PATH
RULE HI_PATH -> {0.25,0.75}, (HI_PATH + HI_PATH) | (hi_path)
RULE LO_PATH -> {0.10,0.60,0.30}, (LO_PATH + LO_PATH) | (lo_path + pipes + lo_path) | (lo_path)

#START variable name
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 small puzzles that lend variety to the randomized levels and serve a dual purpose in allowing Adaptive Mario to fit the player's preferred style.

First, by providing ever more difficulty scenarios, challenge is maintained. This trend also prevents a feedback loop wherein a player jumps frequently because a level contains many platforms, which causes the next level to contain many jumping puzzles and so on.

Second, specific Challenges can be chosen to allow Adaptive Mario to discriminate between play styles when a player's metrics are borderline between two typical profiles. Thus a player who is both a Runner and a Jumper could be given a very difficult jumping challenge to cause the metrics to trend one way or the other. However, this advanced adaptation was not implemented during this iteration of the project.

Evaluation

Several challenges were overcome during the implementation of Adaptive Mario. From a technical standpoint, finding a Java-compatible Rete rule engine (Drools) was a time-consuming effort, mainly due to the large number of third-party JAR dependencies. Sufficient time was invested in this endeavor before integration was ultimately successful that the role of the rules engine was minimized during the design process. Consequently, only basic alterations are made to the random level structure based on the player profile.

Randomized level generation using a stochastic CFG was largely successful, however. Not only can this very simple grammar generate a variety of levels, implementation of a parser for the rule set allows a designer to make meaningful content changes without the need to recompile source code. Such grammars are limited of course - integration between the

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.

top-down macro generation code and bottom-up Challenge Component code was a challenge, as the LevelGrammar module lacks the capacity to plan ahead and avoid awkward juxtapositions of LevelComponents.

While the arbitrary scoring and single-iteration feedback loop of the PlayerProfile module functions as designed, the core PlayerProfile model represents the most promising area of improvement for a future version of Adaptive Mario. Because small changes to the scoring weights can make a great difference in determining which level-building rules are ultimately invoked, better tuning of these parameters is vital. Consequently, this package would benefit greatly from use of more advanced techniques (perhaps Artificial Neural Networks or Bayesian inference) to discover more accurate scoring values.

Conclusion

It seems clear from the 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). One potential issue exists, however - it is possible that some players simply are not fans of the platform jumper genre!

Fortunately, the architecture of Adaptive Mario (and the externalization of the rule engine and level grammar in particular) means that the designer is not limited to the side-scrolling platform jumper palette. It is easy to imagine that Adaptive Mario could be transformed into a Roguelike or adventure game without substantially altering the core Mario mechanics. Just as the player is freed from the constraints of pre-generated content, the designer (or hobbyist) is given the necessary tools to alter the game world as desired.

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`.

seed random number generator seed value (`long`)

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.