AAAI Technical Report WS-11-19

Game Metrics Without Players: Strategies for Understanding Game Artifacts

Mark J. Nelson
Center for Computer Games Research
ITU Copenhagen
Copenhagen, Denmark

Abstract

Game metrics are an approach to understanding games and gameplay by analyzing and visualizing information collected from players in playtests. This paper proposes that another source of metrics is the game itself, and that not all information needs to (or ought to) come from empirical playtests. I discuss seven strategies for extracting information from games, and discuss how the information retrieved in this manner relates to empirical playtest metrics—which it differs from but can often complement.

Introduction

A common (and sensible) way for a game designer to improve his or her understanding of a design-in-progress is to playtest a prototype. Doing so gives the designer empirical information about what players do in the game (and when and how they do it), as well as about their subjective reactions. There has been considerable recent work in using visualization and AI tools to improve the process of collecting and understanding this empirical information. The most well-known visualization is probably the "heatmap", a map of a game level color-coded by how frequently some event occurs in each part of the map, allowing a quick visual representation of, *e.g.*, where players frequently die (Thompson, 2007). This can be extended into more complex analysis of gameplay patterns (Drachen and Canossa, 2009), characterization of play styles (Drachen, Canossa, and Yannakakis, 2009), and analysis of player experience (Pedersen, Togelius, and Yannakakis, 2009).

In all these approaches, the source of information is *exclusively* the player. Empirical information is collected from players, by methods such as logging their playthroughs, tracking their physiological responses during play, administering a post-play survey, etc. Then this data is analyzed and visualized in order to understand the game and the gameplay it produces, with a view towards revising the design.

For some kinds of game-design questions, it's sensible or even necessary for our source of information to be empirical data from players. If we want to know if a target audience finds a game fun, or what proportion of players notice a hidden room, we have them play the game and find out. But an additional purpose of playtesting is for the designer to better understand their *game artifact*, in the sense of a system of code and rules that works in a certain way. Some common results of playtesting aren't really empirical facts at all. When the designer looks over a player's shoulder and remarks, "oops, you weren't supposed to be able to get there without talking to Gandalf first", that isn't an empirical fact about players or gameplay that's being discovered, but a logical fact about how the game works.

While this kind of improved understanding of a game artifact *can* be discovered through playtesting, the only real role of the player in uncovering that kind of information is to put the game through its paces, so the designer can observe it in action. We need not treat the game as a black box only understandable by looking at what happens when players exercise it, though; we can analyze the game itself to determine how it operates. Indeed, designers do so: when they design rule systems and write code, they have mental models of how the game they're designing should work, and spend considerable time mentally tracing through possibilities, carefully working out how rules will interact, and perhaps even building Excel spreadsheets before the game ever sees a playtester. Can we use AI and visualization techniques to augment that thinking-about-the-game-artifact job of the designer, the way we've augmented thinking about player experience?

This paper sketches seven strategies for doing so, several existing and others new. While they can be used as alternatives to player-based metrics and visualizations for some kinds of design questions, especially early on in prototyping (so that the designer can focus playtesting on more subjective or experiential questions), many also work naturally alongside existing metrics/visualization approaches. Although my own work (with collaborators) has been based on modeling game mechanics in symbolic logic (Nelson and Mateas, 2008; Smith, Nelson, and Mateas, 2009, 2010), this paper attempts to discuss strategies in a way that's open to a range of technical approaches that could be used to realize them—focusing on *what* we might want to get out of analyzing games and why.

Strategy 1: "Is this possible?"

The easiest questions to ask are of the form: can X happen? Examples: *Is the game winnable? Can the player collect ev-*

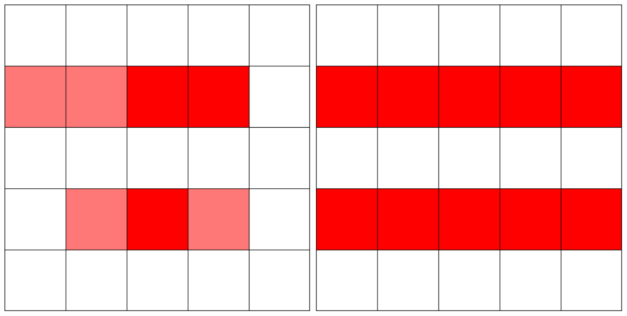

Figure 1: Heatmaps of player deaths: left is empirical deaths from several playthroughs, while right is analytically possible death locations.

ery item? Can the player die while still in the first room? Can the player die right on this spot I'm clicking? Can both of these doors ever be locked at the same time?

This strategy answers any yes/no question whose answer is determined directly by the rules of the game: whether a game state is possible, or an event can ever happen, given all possible player behaviors. In fact questions might not even involve variation on the player's part, but variation in the system: *Do enemies ever spawn at the bottom of a pit?*

This analysis strategy could be exposed directly to the designer in a query setup. Alternately, classes of answers can be visualized. For example, we can ask, for every square in a grid-divided level, whether the player could die there, and color-code the map accordingly, producing a version of the popular player-death heatmap that shows instead where it is *possible* for players to die.

Figure 1 shows on the left, a heatmap of deaths in a few playthroughs of a room in a Zelda-like game; and on the right, a map of where it's possible to die. There's a clear structural pattern immediately visible in the second figure, derived from the game rules rather than from empirical playtest data: the player can only die in two specific rows. In the first figure, this pattern hasn't quite been made clear via the pattern of empirical player deaths. Especially with more complex patterns, it can take a lot of playtesting data to spot these sorts of structural features in the possibility space, which are usually caused by unnoticed interactions of rules, or interactions between game mechanics and level design. In addition, it can be useful to have both kinds of heatmaps, to allow the designer to disentangle which patterns are caused by the possibility space, and which are caused by patterns within player behavior. A figure like the second one can even be used to tweak level design at a more detailed level, *e.g.* to place safe spots.

Several techniques can been used to implement this strategy. Salge et al. (2008) playtest games with a simulated player that evolves itself in order to try to achieve particular outcomes. My own work uses logical inference for query-answering (Nelson and Mateas, 2008). More specific (and likely more efficient) algorithms can be used for special cases of possibility as well; for example, flood-fill algorithms are sometimes used to make sure there are no disconnected parts of a level, and graph-reachability algorithms can be used for similar purposes. A challenge with using these special-case algorithms in a larger system is to automatically recognize when they're applicable, and in which variants; for example, basic flood-fill suffices for reachability as long as a level doesn't involve movable walls or keys.

```
happens(move(player,north),1)
happens(attack(monster,player),2)
happens(attack(monster,player),3)
happens(die(player),3)
```

Figure 2: Gameplay trace of a player dying.

Strategy 2: "How is this possible?"

Beyond finding *that* something is possible, a designer often wants to know *how* it could happen. In player metrics this is answered by collecting a log or trace of the actions the player took, along with information about game state (such as the player's position and health). These traces can be used both for debugging (to track down how something happened when it shouldn't have been possible), and as a way to understand the dynamics of the game world, by looking at the paths that can lead to various game states.

Figure 2 shows a fairly boring event log of a player dying after walking up to a monster and then doing nothing. This could be elaborated with the value of various game states at each point in the log, such as the player's position and health. However, its boringness of course raises the follow-on question: can you show me not only ways that something is possible, but *interesting* ways that it could happen? For some outcomes, like those that should never happen, any log is interesting, but for others this is a trickier question. One approach is to let the designer interactively refine the trace they're requesting. In this example, they could ask for a way the player dies *without* ever standing around doing nothing, and then go on to add more caveats if the result were still too mundane; we refer to this as trace zooming (Smith, Nelson, and Mateas, 2009).

In a simulation framework like that of Salge et al. (2008), the log of how something is possible would simply be a log of the actions taken during the successful simulation run (although it may take time to recompute new runs if something like interactive zooming is offered). In a logical framework, it can be posed as an abduction problem: finding a sequence of events that, if they happened, would explain how the sought-after outcome could come about (Nelson and Mateas, 2008); zooming would be abduction with added constraints on the explanations. It's also possible to view finding a path to an outcome as a planning problem within the story world, and use a classical AI planner. For example, Pizzi et al. (2008) find solutions to game levels and display them as comic-like sequences of illustrated events.

Strategy 3: Necessity and dependencies

Once we know what things are possible, and how they can happen, we might also want to know what *must* happen. Can you beat *Final Fantasy VI* without ever casting "Meteo"?

Which quests can the player skip? Is this powerful sword I just added to the game needed or superfluous?

These kinds of questions also relate to the questions that can be asked via the first two strategies. Some kinds of necessity questions can be rephrased in terms of whether it's possible to reach a particular state that *lacks* the property we want to determine the necessity of. For example, whether it's necessary to level-up to level 5 before reaching the second boss is equivalent to asking whether it's possible to reach that boss while at a level of 4 or below. Other kinds of necessity questions can be rephrased in terms of zooming in on traces. For example, asking whether a particular sword is necessary to beat the game is equivalent to asking for a gameplay trace where the player beats the game, which doesn't contain the pick-up-that-sword event.

In empirical playtesting, it's common to collect metrics about usage: how many players achieve a particular quest, use each item, etc. Similarly to how we can juxtapose empirical data with analytically determined possibilities in Strategy 1, in this strategy we can juxtapose empirical data with analytically determined necessities. Of course, if the empirical results show less than 100% for some item or event, it couldn't have been necessary, but on the other hand there may be things that 100% of our playtesters did which *aren't* actually necessary, which this game-analysis strategy would distinguish.

More automatic dependency analysis is also possible. For example, asking whether it's necessary for event A to precede event B, or vice versa, can let us build up a graph of necessary event ordering, which at a glance indicates some of the causal structure of the game world. That includes causal structure that wasn't explicitly written; for example, entering a locked room might have several explicit preconditions, like the player needing to find a key before entering, but also several implicit preconditions caused by interaction of other rules, like the player needing to find a particular suit of armor before entering (because there is simply no way they can successfully get to the room without having first acquired that suit of armor).

To our knowledge, no existing game-analysis work explicitly aims at this kind of automatic necessity or dependency analysis. Our own work follows the approach, described in this section, of reducing necessity and dependency analysis to a series of queries implemented using strategies 1 and 2.

Strategy 4: Thresholds

Sometimes there are magic numbers delineating the boundaries of possible behavior, or of a certain regime of game behavior. What is the shortest possible time to complete a *Super Mario Bros.* level? What range of money could a *SimCity* player possess when five minutes into the game?

This strategy can give useful information often not discovered in initial playtesting, for example by finding speedruns of a level, or cheats to quickly finish it, that the usually not-yet-expert-at-the-game players in a playtesting session wouldn't have found. In addition, it can be paired profitably with empirical player data to give an idea of how close the particular range of data being observed comes to the theoretical bounds that the game's rules define. For example, any graph that graphs players as a distribution on a numerical scale could also draw bounds of the largest and smallest possible value—and not just largest or smallest *a priori*, as in a score that's defined to range from 0 to 100, but actually possible in a specific context.

In addition to telling us whether playthroughs significantly different on a particular metric's axis from the empirically observed ones are possible, the relationship between the empirical range of data and the theoretical extrema can tell us something about the players in our playtest. For example, in earlier work we discovered that the typical players in our playtest of an underground-mining game were much more cautious with returning to the surface to refuel than was strictly necessary (Smith, Nelson, and Mateas, 2009). This can then be used in concert with Strategy 2 to figure out how to achieve the threshold values.

In our logic-based approach, thresholds are found using a branch-and-bound method. A possible solution (of any value) is first found, and then a constraint is added that a new solution be better than the one already found (*e.g.* shorter, if we're looking for the shortest playthrough). Then we look for another solution meeting the new constraints, and repeat until no additional solutions are found. This has the advantage of generality, but can be slow.

Future work on the problem could explicitly use an optimization method, whether a randomized one like genetic algorithms, or a mathematical one like linear programming. In addition, there are a wide range of algorithms to find maximal or minimal solutions to more specific problems. For example, given a model of a level, we can find the shortest spatial path through the level using a standard algorithm like Dijkstra's algorithm. As with the specialized algorithms in Strategy 1, a difficulty in using these specialized algorithms would be automatically determining when they're applicable; for example, the shortest spatial path through a level may not be the shortest actually achievable path, given the game mechanics—it might not even be a lower bound, if the mechanics include teleportation. On the other hand, these kinds of differences might also give information; the difference between the shortest spatial path through a level and the shortest path that a player could possibly achieve through a level might give an indication of its worst-case difficulty, for example, since it would mean that there is some minimal level of off-perfect-path movement the player would have to perform.

Strategy 5: State-space characterization

The strategies so far can be seen as trying to probe a game's state-space from various perspectives. Could we more directly analyze and characterize the state-space of a game?

One possibility is to try to visualize the state space. In any nontrivial game, the full branching state graph will be unreasonably large to display outright. However, it may be possible to cluster or collapse the possible dynamics into a smaller representation meaningful states and dynamics (Cohen, Davis, and Warwick, 2000). In addition, techniques developed for summarizing the state space of empirical player data could be applied to summarizing sampled traces from the overall space of possible playthroughs (Andersen et al., 2010).

More interactively, single traces can display some information about their neighbors in the state space; for example, branch points might show alternate events that could have happened at a given point in the trace, besides the event that actually happened in that trace. This can be used to "surf" the space of possible playthroughs, in a more exploratory manner than the specifically requested traces returned from Strategy 2. Alternately, we can start with empirically collected traces and observe their possibility-space neighbors; this can be used to bootstrap a small amount of playtesting data into a larger amount of exploration, by showing possible—but not actually observed—gameplay that is similar to the observed gameplay.

Moving into more mathematical territory, games largely based on mathematical approaches such as differential equations, influence maps, and dynamical systems (Mark, 2009) might be analyzed using standard mathematical approaches, such as finding fixed points or attractors or displaying phase-space diagrams. Some basic experimentation along these lines is sometimes done during Excel prototyping, but this area (to my knowledge) remains largely unexplored.

Strategy 6: Hypothetical player-testing

The strategies so far try to investigate the overall way a game operates. We could restrict this by trying to characterize only how a game operates with a particular, perhaps highly simplified, model of a player. Such a restriction is not intended mainly to insert a *realistic* player model, but to investigate how the game operates in various extreme or idealized cases. For example, what happens when the game is played by a player who always attacks, except heals when low on health? If that player does very well, the game might be a bit too simple. Or, in a multiplayer game, different players could be pitted against each other to see how they fare, which might tell us something about the design space.

The Machinations system (Dormans, 2009) simulates hypothetical players playing a Petri net game model, and collects outcomes after a number of runs. Our logic-based system applies strategies 1–4 conditioned on a player model, answering various questions about possibility, necessity, etc., under the added assumption that the player is acting in a particular manner. In a slightly different formulation, Monte-Carlo "rollouts" of boardgames pit two possible strategies against each other in a specific point in the game, to determine how they fare against each other (Tesauro and Galperin, 1996).

Strategy 7: Player discovery

While hypothetical players can be useful for probing how a game behaves under various kinds of gameplay, we found that designers often had difficulty inventing such hypothetical players, and instead wanted the process to work backwards: given a game prototype, could we automatically derive a simple player model that can consistently achieve certain outcomes (Nelson and Mateas, 2009)? For example, rather than having to try out questions such as, "can this game be beaten by just mashing *attack* repeatedly?", some designers would prefer we analyze the game and come back with: here is the simplest player model that consistently beats your game.

That question can be seen as a stronger or more generalized version of trace-finding (Strategy 2). Finding how a particular outcome is possible returns *one* possible instance where it could happen. Finding a player that can consistently make the outcome happen is a compressed description of many such instances.

There are several ways to invent these kinds of player models. One approach is to sample many possible traces reaching the requested state (using techniques from Strategy 2), and then inductively extract a player model from these traces, or perhaps several player models from different clusters of traces. There are already techniques from empirical gameplay metrics that can be used to infer such player models (Drachen, Canossa, and Yannakakis, 2009), which could be applied to extracted gameplay traces instead.

A different approach is to directly infer whether there exists a player model from a class of simplified players that can reach the desired state. For example: is there any single button a player can mash constantly to beat the game? If not, is there a 2-state button-mashing finite state machine that can consistently beat the game (perhaps alternating between "attack" and "heal")? If not, we can query for state machines with more states, or other kinds of more complex player models. One axis of complexity is how "blind" the player model is: the button-mashing or alternate-between-two-states model ignores the game state completely. If there's no simple blind finite state machine that can beat the game, how about one that only looks at one game state (or two game states)? If a player model of *that* kind exists, it would tell us something about the game state that is relevant for decision-making. We've been performing some experiments in this kind of player-model inference using logical abduction, but overall the space of possible approaches is quite open.

In multiplayer games, player discovery can be related to game-theory terminology, such as finding optimal strategies (for various kinds of optimality), dominated strategies, etc. While using game theory for videogame analysis or balancing has often been discussed, it seems to resist practical application in part due to the mismatch in scale between the size of games that game-theory software typically handles, and even small videogame prototypes. In particular, most computational game theory assumes either a one-step game, or an iterated (staged) game with relatively few steps, typically as few as three or four; whereas most videogames go on for many timesteps, and have their dynamics emerge over at least slightly longer timescales. Overcoming this problem would require either finding a way to pose many of the interesting problems in terms of less-iterated game-theory problems, or else scaling the tools to many more iterations.

Finally, a large class of gameplay algorithms can be used as player-discovery algorithms of a sort, especially if they produce interesting internal structure that tells us something about the game they learn to play. For example, a reinforcement-learning algorithm that learns a state-value function would be an interesting source of data for augmenting the kinds of state diagrams in Strategy 5, adding to them information about how valuable each state is from the perspective of reaching a particular goal.

Conclusion

While playtesting games is, and will remain, a valuable source of information about gameplay, the game artifact itself need not be a black box, since we can learn many things about a game design simply by better understanding the game itself—how the rules and code inside the game operate and structure play experience. This analysis of games can benefit from an ecosystem of metrics and visualization techniques that will hopefully grow as rich as that now featured in player metrics research.

Towards that end, this paper sketches seven strategies for extracting knowledge from a game artifact: what information we might extract from a game, why we would want to extract it, and how it relates to the kinds of information we can find in playtests. While I've briefly mentioned existing work that tackles the implementation of some of these strategies, much remains unstudied, both in terms of undertaking many kinds of analysis at all, and for those that have been undertaken, in understanding the strengths and weaknesses of various technical approaches to modeling games and extracting information from them.

References

Andersen, E.; Liu, Y.-E.; Apter, E.; Boucher-Genesse, F.; and Popović, Z. 2010. Gameplay analysis through state projection. In *Proceedings of the 5th International Conference on the Foundations of Digital Games (FDG).*

Cohen, P. R.; Davis, J. A.; and Warwick, J. L. 2000. Dynamic visualization of battle simulations. Technical Report 00-16, University of Massachusetts Computer Science Department.

Dormans, J. 2009. Machinations: Elemental feedback patterns for game design. In *GAMEON-NA 2009: 5th International North American Conference on Intelligent Games and Simulation*, 33–40.

Drachen, A., and Canossa, A. 2009. Analyzing spatial user behavior in computer games using geographic information systems. In *Proceedings of the 13th International MindTrek Conference*, 182–189.

Drachen, A.; Canossa, A.; and Yannakakis, G. N. 2009. Player modeling using self-organization in Tomb Raider: Underworld. In *Proceedings of the 5th International Conference on Computational Intelligence and Games (CIG).*

Mark, D. 2009. *Behavioral Mathematics for Game AI.* Course Technology PTR.

Nelson, M. J., and Mateas, M. 2008. Recombinable game mechanics for automated design support. In *Proceedings of the 4th Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE)*, 84–89.

Nelson, M. J., and Mateas, M. 2009. A requirements analysis for videogame design support tools. In *Proceedings of the 4th International Conference on the Foundations of Digital Games (FDG).*

Pedersen, C.; Togelius, J.; and Yannakakis, G. N. 2009. Modeling player experience for content creation. *IEEE Transactions on Computational Intelligence and AI in Games* 1(2):121–133.

Pizzi, D.; Cavazza, M.; Whittaker, A.; and Lugrin, J.-L. 2008. Automatic generation of game level solutions as storyboards. In *Proceedings of the 4th Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE)*, 96–101.

Salge, C.; Lipski, C.; Mahlmann, T.; and Mathiak, B. 2008. Using genetically optimized artificial intelligence to improve gameplaying fun for strategical games. In *Proceedings of the Sandbox 2008 ACM SIGGRAPH Videogame Symposium*, 7–14.

Smith, A. M.; Nelson, M. J.; and Mateas, M. 2009. Computational support for play testing game sketches. In *Proceedings of the 5th International Conference on the Foundations of Digital Games (FDG)*, 167–172.

Smith, A. M.; Nelson, M. J.; and Mateas, M. 2010. Ludocore: A logical game engine for modeling videogames. In *Proceedings of the 2010 IEEE Conference on Computational Intelligence and Games (CIG).*

Tesauro, G., and Galperin, G. R. 1996. On-line policy improvement using Monte-Carlo search. In *Advances in Neural Information Processing Systems (NIPS)*, 1068–1074.

Thompson, C. 2007. Halo 3: How Microsoft Labs invented a new science of play. *Wired.* August 8, 2007. http://www.wired.com/gaming/virtualworlds/magazine/15-09/ff˙halo.