Open Problem: Reusable Gameplay Trace Samplers

Adam M. Smith
Center for Game Science
Department of Computer Science & Engineering, University of Washington
amsmith@cs.washington.edu

Abstract

We identify an open problem in game design assistance and automation: the development of reusable gameplay trace samplers. Inside many sophisticated content generators and design tools is a component that samples interesting and plausible sequences of player actions. Details and summary properties of these samples are used to assess generated content and to inform designers. As the development of this component is technically involved (sometimes comparable to making a second implementation of a game's mechanics), design tools often either make use of entirely custom, game-specific samplers or make due without the ability to sample interesting traces at all. This severely limits the population who could benefit from automation to those who are motivated to develop it for themselves. We propose the development of reusable samplers to ease the development of future design automation tools. This paper reviews several systems that demonstrate the availability of technology required by these samplers and the range of applications they may serve. It also sketches how future samplers might be architected. This proposal identifies one way for technical research to make progress on design automation challenges without making problematic assumptions about the nature of player behavior or designer intent. Filling in this missing infrastructure, we claim, will make the use of artificial intelligence in the design process more accessible and thus accelerate game design projects.

Introduction

The ability to sample representative and interesting gameplay traces is essential to the function of many game content generators. Samples of interesting hypothetical sequences of player actions capture information about the interactive space of play afforded by content artifacts. This same ability powers prototyping tools that intend to give exploratory game designers insightful feedback on the choices they are considering. Building reusable sampling engines would accelerate efforts to develop both content generation and other design automation systems by reducing the required expertise and engineering cost associated with producing accurate and responsive generative simulators for gameplay.

Gameplay trace samplers, as we tentatively call them for the purposes of this proposal, would function as an oracle that stands in for the player. The sampler should be able to provide the concrete player-action data that allows answering higher-level questions such as "how long does it take the player to get from here to there?" "what actions might the player take to recover from this situation?" or "how often will players who play according to this externally provided strategy get stuck at each point in the level?" The ability to sample such traces would allow asking and answering questions that are not just about actions, but also about the existence of traces with critical properties for a generated map or puzzle, or even a designer's manually entered rule system or strategy description.

As an introductory example of the use of gameplay sampling in an active design project, consider an in-house tool created to assist in the development of the upcoming game *The Witness*.[1] In a post on the game's development blog, developer Casey Muratori describes Walk Monster, a system for mapping out the player-traversable area in the game's map. Muratori writes (2012):

> There are very easy ways to think that you were testing the player movement code without actually testing it. One example would be to do some kind of analysis on the collision volumes and walkable surfaces in the game, looking for small areas, gaps, etc. Once you'd eliminated all of these, you'd then proclaim the world safe to traverse and move on. But this is testing the data, not the actual code. It's still easily possible to have bugs in the movement code that result in bad behavior even with sanitized data. To avoid this kind of trap, I wanted the testing system to remain as close as possible to what a human does to actually control player movement in the game.

His article describes a succession of sampling algorithms that lead to quicker and more informative feedback about what was possible in the game (towards catching design flaws earlier in the development process). He explains how he abstracts out physical momentum to focus on the interaction between the map geometry and the game's collision detection logic. This choice mildly sacrificed accuracy of his

[1] http://the-witness.net/

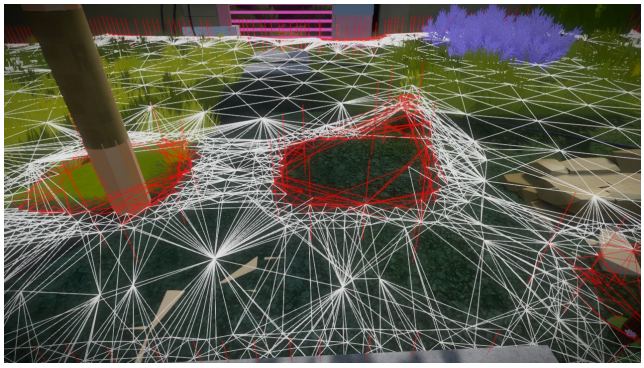

Figure 1: A custom sampling algorithm for traversable spatial paths in *The Witness* reveals a design bug. The red hole in the center outlines an invisible collision body mistakenly left on the map. (This screenshot was reproduced from Muratori's blog post with permission from the developers.)

sampler in favor of reduced time to gather the sampling coverage he sought, a balancing choice any sampling system architect must make. In use, the system identified that a particular patch of rocks was traversable by the player, despite the intent for them to be an impassible barrier. Movement samples found by the tool (and later reproduced in live gameplay) further revealed this this patch was only traversable in one direction, potentially allowing the player to become stuck. In another instance (visualized in Figure 1), the lack of traces crossing one area of the map highlighted the fact that a collision body had been left there after its graphical counterpart had been removed.

Even with the need for original algorithm design, Muratori is convinced that his effort in making and applying the sampler were worth the effort. In his closing thoughts, he ponders what other design queries might also be served by moderate extensions to his experiment. It is unrealistic to ask every other developer to take the blind leap into sampler design in the way that this developer did, but, with reusable infrastructure, others might be persuaded to try and see what benefits can be had for their own projects.

In the body of this proposal, we review several existing systems that internally perform functions similar to our proposed samplers and demonstrate potentially reusable technology for developing them. We also sketch possible architectures for samplers in terms of their inputs, internal processes, outputs, and possible modes of packaging for use in specific game design projects. Finally, we examine the potential impact of the proposal on technical research in and outside of game content generation.

Samplers in the Wild

The seeds of potential reusable gameplay trace samplers are scattered in two directions: in game-specific samplers that could be generalized to talk about other games in their target genres, and in already-general sampling tools that can be made more specific for improved accessibility in game development projects.

In Game-Specific Projects

Game-specific tools can take advantage of sampling techniques that would not be appropriate to blindly apply to every game. In the assisted level design tools (Bauer and Popović 2012; Bauer, Cooper, and Popović 2013) for the game *Treefrog Treasure*,[2] two internal gameplay trace samplers are used to determine reachability between different points in a level design under construction. An initial sampler, based on rapidly-exploring random trees or RRTs[3] (LaValle and Kuffner 2001), determines overall reachability between different platforms in the game's map. After each local edit to the map, a custom dense sampling process reestimates the proportion of moves that could be made between each pair of platforms of interest. While the first version of this system made calls to the actual game to perform simulated actions (resulting in an ability to sample on the order of five moves per second), the subsequent version used genre-specific knowledge about the shape of jump paths to replace a frame-by-frame physics simulation with a jump-at-a-time simulator based on intersecting analytic curve segments (allowing hundreds of simulated moves per second). In a related performance-oriented prototype,[4] careful attention to the sampling process allowed sampling hundreds of thousands of moves per second for the same movement mechanics. In general, there are many ways to trade small decreases in simulation accuracy for massive increases in sampling rate, but the scope of these approximations is limited to games with similarly structured mechanics.

In another project from the same group, Butler et al. (2013) provide interactive feedback on manual edits as part of a mixed-initiative design tool for the puzzle game *Refraction*.[5] Their level analyzer works by sampling solution traces that avoid practicing critical gameplay concepts. The presence or absence of these solutions feeds into a visualization of which concepts a puzzle requires across all of its solutions. Although this sampler depends on game-specific code (approximately 150 lines of code that define the game's mechanics and solution conditions), the engine that searches for satisfying solutions is quite general (Smith et al. 2012). The same idea is used in their level generator which makes use of exhaustive machine playtesting internally (Smith, Butler, and Popović 2013).

As *Refraction* is a turn-based puzzle game with deterministic mechanics played on a coarse grid, extracting out just those mechanics relevant for efficient and informative sampling is not as hard as it is for other games. In Togelius' personalized racetrack generator (2007), just a single gameplay trace (for a player-specific driver model) is sampled for each track considered by the generator because of the frame-by-frame physics involved in the game. If a probabilistic driver model were to be used (allowing the system to know that a driver could only make a critical turn 20% of the time),

[2] `http://www.kongregate.com/games/gamescience/treefrog-treasure`

[3] Muratori also explored using RRTs in Walk Monster.

[4] `http://adamsmith.as/typ0/k/frogsplat/`

[5] `http://www.kongregate.com/games/gamescience/refraction`

many samples would need to be taken, either putting pressure on the sampler to perform much faster or the generator to glean the same information from traces sampled with a coarser physics timestep. In the generator for *Cloudberry Kingdom*, a procedural platformer game hyped for its "insane" levels (Fisher 2012), generated levels are guaranteed to be feasible (possible to complete) by constructing a reference path. After each new design element is added to the level, such as platform or fireball, the reference trace must be re-simulated to check the path's feasibility. Game-specific techniques could be invented to speed up sampling for either of these games, but unless they are shared and made reusable, it is only the daring sampler developers who will rediscover them in new projects.

In General Game Design Tools

General-purpose gameplay trace sampling is far from impossible. Game prototyping systems such as LUDOCORE (Smith, Nelson, and Mateas 2010) and the Machinations framework (Dormans 2011) offer trace sampling for any game representable in their abstract and diagrammatic modeling languages. In LUDOCORE, a model of the game's mechanics, world configuration, player behavior, and context of designer interest are encoded in a common logical modeling language. In response to these heavyweight queries, the system produces one or more example gameplay traces demonstrating the required (and potentially extremely rare) behavior. No statistical properties, however, are promised over the output. The Machinations framework, by contrast, takes a lightweight approach where samples are constructed by independent forward-simulation runs. To sample specific types of traces (e.g. to explore a certain strategy), imperative scripts can be written to conditionally trigger events in each sampled run.

The presence and probabilistic distribution of gameplay traces with interesting properties can directly translate into design insight regarding a design under test. In LUDI (Browne and Maire 2010), "a system for playing, measuring, and synthesizing games," an estimate of the quality of a given strategic board game is made by taking many samples of its typical gameplay (as generated by generic strategic search agents). Whether the distribution of samples has properties such as bias for or against the first player, too often ending in a draw, or never indicating the possibility of dramatic reversals partially reflects the nature of the game. However, properties of these samples also reflect the nature of assumptions the system made about how typical players play.

To gain deeper insight into which facets of a game design are responsible for its balance, Jaffe et al. (2012) argue for examining the outcome of competitive play under designer-prescribed restrictions on the behavior of one player or another. For example, to judge the impact of a particular action, they examine the relative win rate of a player who can only use that action up to a constant number of times against an unrestricted opponent. Although the system they describe is based on exhaustive analysis of the complete game tree, systems that attempt to provide the same insight for more expansive game trees (such as those derived from complex videogames) will come to depend on sampling techniques. Jaffe's later work examined the possibility of Monte Carlo tree search (MCTS) for this purpose (2013). If efficient and reusable samplers are available for a variety of game types, many interesting design questions can be automatically probed. To do this, the designer poses and interprets the results of queries for gameplay traces under restriction on play. In this application, the sampler does not judge what is good or bad design, but it does provide an "early warning system" for notable imbalances.

Automatic navigation mesh construction (Tozour 2002) is an example of a reusable designer assisting technology that has already been widely adopted within the game industry. These meshes, however, are most often computed directly from the geometry of a level design. As Muratori reminds us, this data-only analysis misses interactions between the game's varied mechanics and the fixed geometry. Reusable gameplay trace samplers might someday allow the automatic construction of more general 'action meshes' which captured not just possible navigation actions but engagement with other mechanics such as triggering of switches, unlocking doors, collecting/crafting/using items, or dialog exchanges. The boundaries between spatial pathfinding, action planning, and abstract puzzle solving would become blurred in such systems.

Architectural Choices

In this section, we sketch the architecture of potential samplers. Depending on the choice of inputs, internal processes, and outputs, a trace sampler might be better named by other terms such as trace simulator, trace planner, or trace constraint solver. In each of the architectural facets below, a satisfying system need not implement every possibility, instead only a combination that works well together.

Inputs

A sampler's input defines the game in question and poses a specific query to focus the samplers attention. Thus, the sampler's input should contain details such as the level design in question and the initial state of the player and non-player characters. Similarly, the input should describe the relevant victory or loss conditions associated with the scenario (such as reaching a key location within allotted time or touching objects of a certain type). The sampler will not always be asked to find traces resulting in the victory or loss conditions, but telling the sampler when a trace would result in termination of a gameplay session is part of how the game is defined.

Beyond defining a session's initial and final conditions, the sampler should take a description of the specific mechanics of the game (either as a symbolic description or direct callback mechanism). Within a given genre, many low-level mechanics may be assumed, and a smaller collection of rules and parameters can define the specifics of a game in terms of the same abstractions afforded by the game engine on top of which it is built. The granularity of simulation used by the sampler need not exactly match that of the game in question. Abstract simulations, for example, could be used to answer

tentative design questions for early-stage game designs before the fine-grained details have been fully worked out.

The above inputs would be sufficient for a sampler to draw from a default distribution over all play traces (perhaps taking every available action with uniform probability). In order to sample more interesting traces, the sampler should also take a description of the expected player behavior. This could be provided as either an explicit distribution over actions as a function of game state (perhaps derived from previously collected observations of human players) or as a callback to a system outside of the sampler to get a distribution over moves. This information would ensure that traces emerging from the sampler are realistic, but they might still be irrelevant to the design concern in question. Thus, the sampler should also take an optional description of any other property that should be required of the sampled traces. This might be the constraint that the traces all end in a certain state, that actions of a certain type are never used (despite the provided player model suggesting their preference), or that conditions outside of the player's control are structured in a certain way (perhaps that enemies should be assumed to always fail their first attack). Whether this information is provided as explicit constraints or as a function over game state and player actions that should be optimized is a choice left for the sampler's architect.

Processes

Three broad directions for the internal processes used by the sampler are particularly clear to us. In the first direction, the sampler would be organized as a statistically-sound sampling process. Such a system would draw samples with the guarantee that the sample distribution converges to a well-defined theoretical distribution (associated with the query to the sampler). Probabilistic programming languages like Church (Goodman et al. 2008) offer these semantics for a Scheme-like general-purpose programming language. These general-purpose languages fall short of desirable, for our application, by providing no special support for creating and refining models of gameplay for any particular game genre. Nevertheless, such a language could be used inside of a sampler with the help of a library that forms, in effect, a probabilistic game engine for the genre in question.

The second direction, also inspired by the availability of tools for analyzing general-purpose programs, is to use tools emerging from the software model checking community. PRISM (Kwiatkowska, Norman, and Parker 2011), for example, is a probabilistic model checker that accepts a model of a program and a formal property to check, such as whether a certain class of states is reachable from the initial conditions. PRISM and similar systems can output either detailed instances (interpretable as execution traces) that falsify a critical property or report summary statistics on those traces satisfying the property. Meanwhile, CBMC (Clarke, Kroening, and Lerda 2004) is a model checker that operates on programs defined by standard ANSI-C and C++ sources (potentially allowing exact symbolic inference about the functioning of the same code used by the game in question without directly executing it in a callback, similar to LUDOCORE).

The third direction is to employ general game tree search methods such as MCTS (Tavener et al. 2012). These are the same kind of algorithms that power the various agents that participate in general game playing (GGP) competitions (Genesereth, Love, and Pell 2005). The trace-sampling problem, however, will not always be aligned with the strengths of common tree search algorithms. For example, the relevance of a trace (deciding if the sampler should emit the trace) may not be representable as a simple reward function over states. Instead, it may depend on properties of the sequence of actions taken. As a result, different internal sampler processes may be better suited to answering different types of queries.

Outputs

Depending on the level of detail used by the gameplay simulation and the scope of the query, different samplers may intentionally provide outputs at different granularities. For example, one sampler may only report high-level moves in an approximation of the game, perhaps under the extra constraint that each sample is unique and the collection of samples exhausts the possibilities of a given size. This might be used to enumerate build orders in an RTS game that, under abstraction, result in a sufficiently strong economy within a certain time limit. Alternatively, a sampler that operates over a very accurate model of the game's mechanics might report pixel and/or frame-accurate traces that can be replayed exactly within the target game.

It might be tempting to directly output aggregate statistics or other metrics over possible traces in an effort to directly answer a design-level question. We suggest that architects of samplers focus on emitting informative and concrete gameplay traces wherever possible. This will allow a larger system employing the sampler to draw its own inferences (even if it means simply tracking the counts of traces with certain properties or plotting their extent on a map) and avoid feature creep in the design of the sampler itself that might limit its reusability. The sampler's job is to draw samples, not to interpret those samples on behalf of others.

Packaging

Reusable samplers become useful only when integrated into a project-specific context. Perhaps the simplest way to package a sampler would be as a library that provides a modeling and query API (with an eye towards direct integration into larger tools and designer-facing visual interfaces). This approach allows for easy integration of hooks and callbacks. Another approach is to package the system as a network-accessible service with a well-defined query language (akin to a database engine). This language-agnostic approach would be a convenient way to expose cluster and cloud computing resources for use in design automation purposes while shielding consumers of the sampler from the associated system complexity. Yet another alternative would be providing a domain specific programming language and interpreter (where programs define the game model and query, and, when run, produce a result set of samples). This self-contained approach might be best useful for very exploratory design processes where performance and integra-

tion are not yet serious concerns while keeping the rapidly changing game model in one place is preferred.

Discussion

Having sketched the array of existing samplers and explored potential architectural choices for reusable samplers in the future, we now look at the effect of sampler development on technical game research programs.

Problematic Assumptions

Returning to the context of design automation, the development of reusable samplers would provide a productive outlet for technical research in a way that avoids making problematic assumptions. It is common for game content generators (such as level generators) to make explicit and architecturally fixed assumptions about what makes for good game design or what the imagined designer (the user) always wants from the generated content. Inside these systems, the component responsible for generating representative gameplay traces for evaluation makes similarly problematic assumptions about how players behave: that just one trace or a very sparse collection of traces is representative of the target audience (such as in *Cloudberry Kingdom*), or that the distribution over traces emerging from the system's fixed search process is always representative (such as in Ludi). Smith et al. examine the design implications of undersampling in more detail (2013).

Where possible, the criteria for what makes a gameplay trace interesting should be taken as an input (if they can be declaratively described) or made available to the sampler via hooks or callbacks to project-specific systems outside of the sampler instead of being fixed internally. The intent is to allow a designer to ask new questions on a whim, without re-engineering the sampler, or allowing an enclosing system, such as a game-specific content generator, to treat the sampler as a trusted and informative oracle for the trace-sampling problem.

In content generation systems, often a mix of easy-to-formalize gameplay properties (e.g. reachability in interactive play) are combined with hazily defined properties (e.g. visual aesthetics). Instead of treating both types of concerns uniformly in a generator's search process, reusable gameplay trace samplers could provide clear and objective evidence to judge the formalizable properties of the content artifact. This would decouple the design of the search process that is responsible for exploring an artifact's gameplay from the search process responsible for finding good artifacts (where guarantees about the semantics of outputs are much more informative for the inner search process than the outer process). That a larger design automation system deals with informal properties need not imply that the sampling subsystem works on an equally informal basis.

Technical Expertise & Engineering Costs

In the ideal case, technical work put into implementing reusable samplers displaces equivalent work that would have been required to produce a project-specific sampler in larger projects requiring one. Here, we look at look at the potential costs associated with developing samplers as a way to both estimate what effort can be saved and what effort must be expended to produce these samplers.

Reusability demands some degree of generality in how games are represented for or exposed to the sampler (though generality across genres is not an immediate concern). Even in the context of a single game design project, the ability to reuse a sampler across several design iterations would be desirable. The developers of a sampler are responsible for designing representations for game mechanics, player behavior, and other conditions on traces that balances the interests of generality with immediate usefulness on design projects in a specific genre. Game engine developers are in a good position to make some of these choices on behalf of the developers who use their engine (in the same way that a game engine might support a single type of navigation mesh for all client games).

Advanced state-space and symbolic model checking tools internally make use of state-of-the-art combinatorial search algorithms (Edelkamp et al. 2008). The technical expertise required to reliably implement such advanced techniques is unlikely to overlap with the expertise needed to develop rich, genre-specific design tools. Although samplers can provide centralized implementations of powerful but otherwise inaccessible algorithms, some effort will always be needed to translate the abstract outputs of these techniques into game-relevant terms.

As many instances of the gameplay trace sampling are structurally similar to the problem of sampling photon paths in graphics applications (ray tracing), there are significant opportunities for parallelization, approximation, and sophisticated sampling methods. To pick just one example, beam tracing is alternative to ray tracing that, instead of tracking individual rays of infinitesimal width, models thick beams of light which account for bundles of related paths, accounting for whole polygonal surfaces at a time (Heckbert and Hanrahan 1984). In any one application, these advanced implementation techniques might be ignored in the interest of engineering costs, but, in a reusable system, applying these methods become much more attractive.

Extended Reach

If samplers are going to give designers visibility into the broad space of play for the games they support, they need to explore this space significantly faster than the designers can do themselves in traditional playtesting. We conjecture that the quality of insight a sampler can generate is proportional to the logarithm of the number of traces considered by the sampler times a factor related to the level of abstraction used by the sampler. As a result, systems that intend to deliver deep insight for designers (or precise metrics for larger automated systems) will demand that samplers have a wide reach across the space of play arising from a query scenario. This implies, compared to a reference setting of manual playtesting, we should demand a significant increase over real-time simulation. Whether this extended reach should be provided by clever abstraction or aggressive parallelization, however, is a subjective design choice.

New Applications

Absent the availability of samplers, many applications that depend on them, particularly design tools that might rely on them for interactive feedback, cannot be explored. In a platformer level design setting, geometric platforms could snap, not just to a uniform grid, but to the point of first player reachability from other platforms. For post-deployment analysis, it should be possible to sample low-level gameplay details that are consistent with high-level log data, allowing statistical inference over unrecorded properties of human player traces. Being able to sample alternative futures from snapshots of game state observed during high-interest play (e.g. eSports) would allow running detailed what-if scenarios from player models fit to expert play styles. Until such samplers are readily available, we can only speculate on other what new applications might be opened up.

Closing Challenge

The technology required to develop reusable samplers is available, but it needs to be repackaged in a more useful way. In particular, generic sampling procedures need to be demonstrated in a way that makes their usefulness in game design processes clear. Game-specific shortcuts and assumptions should either be generalized or factored out of the design of the samplers to improve reusability. Meanwhile, new applications that were not possible in a world without access to samplers should be explored.

The challenge to developers of these samplers is to make them reusable across game designs so that design exploration (assumed to be contained within a single genre at a time) is cheap to setup and apply. In order to be useful, the interface (likely at the level of APIs) needs to expose enough control that interesting and unforeseen game and designer-specific questions can be asked and answered by sampling specific kinds of traces. Samplers should just sample, accurately, efficiently, and in a programmable direction, leaving summarization and interpretation to project-specific components.

We hope this open problem definition spurs development of technology that changes the conversation around the use of artificial intelligence and other forms of automation in the game design process from "we've never tried something like that before" to "we're looking for a sampler that works for our game's genre."

References

Bauer, A., and Popović, Z. 2012. RRT-based game level analysis, visualization, and visual refinement. In *Proc. Artificial Intelligence and Interactive Digital Entertainment Conf.*

Bauer, A.; Cooper, S.; and Popović, Z. 2013. Automatic redesign of local playspace properties. In *Proc. Foundations of Digital Games Conf.*

Browne, C., and Maire, F. 2010. Evolutionary game design. *Computational Intelligence and AI in Games, IEEE Trans. on* 2(1):1–16.

Butler, E.; Smith, A. M.; Liu, Y.; and Popović, Z. 2013. A mixed-initiative tool for designing level progressions in games. In *Proc. User Interface Software and Technology, ACM Symp. on*.

Clarke, E.; Kroening, D.; and Lerda, F. 2004. A tool for checking ANSI-C programs. In Jensen, K., and Podelski, A., eds., *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 2988 of *LNCS*. Springer Berlin / Heidelberg. 168–176.

Dormans, J. 2011. Simulating mechanics to study emergence in games. In *Workshops at the Seventh Artificial Intelligence and Interactive Digital Entertainment Conf.*

Edelkamp, S.; Schuppan, V.; Bosnacki, D.; Wijs, A.; Fehnker, A.; and Aljazzar, H. 2008. Survey on directed model checking. In *AAAI Wksp. on Model Checking and Artificial Intelligence (MoChArt)*, 65–89.

Fisher, J. 2012. How to make insane, procedural platformer levels. *Gamasutra*. `http://www.gamasutra.com/view/feature/170049/How_to_Make_Insane_Procedural_Platformer_Levels_.php`.

Genesereth, M.; Love, N.; and Pell, B. 2005. General game playing: Overview of the AAAI competition. *AI Magazine* 26(2):62.

Goodman, N. D.; Mansinghka, V. K.; Roy, D. M.; Bonawitz, K.; and Tenenbaum, J. B. 2008. Church: A language for generative models. In *Proc. Conf. on Uncertainty in Artificial Intelligence (UAI2008)*, 220–229.

Heckbert, P. S., and Hanrahan, P. 1984. Beam tracing polygonal objects. In *ACM SIGGRAPH Computer Graphics*, volume 18, 119–127. ACM.

Jaffe, A.; Miller, A.; Andersen, E.; Liu, Y.; Karlin, A.; ; and Popović, Z. 2012. Evaluating competitive game balance with restricted play. In *Proc. Artificial Intelligence and Interactive Digital Entertainment Conf.*, 26–31.

Jaffe, A. 2013. *Understanding Game Balance with Quantitative Methods*. Ph.D. Dissertation, University of Washington.

Kwiatkowska, M.; Norman, G.; and Parker, D. 2011. PRISM 4.0: Verification of probabilistic real-time systems. In Gopalakrishnan, G., and Qadeer, S., eds., *Proc. Conf. on Computer Aided Verification*, volume 6806 of *LNCS*, 585–591. Springer.

LaValle, S. M., and Kuffner, J. J. 2001. Randomized kinodynamic planning. *The International Journal of Robotics Research* 20(5):378–400.

Muratori, C. 2012. Mapping the islands walkable surfaces. `http://the-witness.net/news/2012/12/mapping-the-islands-walkable-surfaces/`.

Smith, A. M.; Andersen, E.; Mateas, M.; and Popović, Z. 2012. A case study of expressively constrainable level design automation tools for a puzzle game. In *Proc. Foundations of Digital Games Conf.*, 156–163. ACM.

Smith, A. M.; Butler, E.; and Popović, Z. 2013. Quantifying over play: Constraining undesirable solutions in puzzle design. In *Proc. Foundations of Digital Games Conf.*

Smith, A. M.; Nelson, M. J.; and Mateas, M. 2010. Ludocore: A logical game engine for modeling videogames. In *Comp. Intel. and Games (CIG), IEEE Conf.*, 91–98.

Tavener, S.; Perez, D.; Samothrakis, S.; and Colton, S. 2012. A survey of monte carlo tree search methods. *Computational Intelligence and AI in Games, IEEE Trans. on*.

Togelius, J.; De Nardi, R.; and Lucas, S. M. 2007. Towards automatic personalised content creation for racing games. In *Computational Intelligence and Games, IEEE Symp.*, 252–259. IEEE.

Tozour, P. 2002. Building a near-optimal navigation mesh. *AI game programming wisdom* 1:171–185.