UNIVERSITY OF CALIFORNIA
SANTA CRUZ

ANALYSIS OF INCREMENTAL DESIGN CHANGES IN VIDEO GAMES WITH AUTOMATIC EXPLORATION

A dissertation submitted in partial satisfaction of the
requirements for the degree of

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

by

Kenneth Chang

June 2022

The Dissertation of Kenneth Chang
is approved:

Assistant Professor Adam M. Smith, Chair

Professor Magy Seif El-Nasr

Associate Professor Peter Alvaro

Peter F. Biehl
Vice Provost and Dean of Graduate Studies

Table of Contents

List of Figures **v**

List of Tables **ix**

Abstract **x**

Dedication **xii**

1 Introduction **1**

2 Background **7**
2.1 How game software development diverges from traditional software development . . . 8
2.2 Academic Research on Software Analysis . . . 11
2.3 Software testing . . . 13
2.3.1 QA Testing . . . 14
2.3.2 Continuous Integration . . . 16
2.4 Videogame development . . . 19
2.4.1 Traditional testing of games . . . 19
2.4.2 Scripted testing of videogames . . . 21
2.5 Machine Playtesting . . . 23
2.6 Automated gameplay . . . 27
2.6.1 Playing games with AI agents to beat humans . . . 27
2.6.2 Playing games with exploration-centric AI agents . . . 28

3 Exploring from a Backbone **31**
3.1 Introduction . . . 32
3.2 Reveal-More Design . . . 34
3.3 Evaluation . . . 37
3.3.1 Quantitative Analysis . . . 39
3.4 Discussion . . . 43

4 Visualizing Incremental Changes **45**
4.1 Visualizing Design Changes with Differentia 47
4.2 Technical Design of Differentia Prototype 49
4.3 Visualization Design 50
4.3.1 Gathering Gameplay Data 51
4.3.2 Representing Game Moments as Vectors 52
4.3.3 Grouping and Arranging Moments 52
4.4 Example Applications 53
4.4.1 Incremental Change: SMW vs GravHack 54
4.4.2 Extensive change: Pokemon Red vs. Brown 55
4.4.3 No change: SMW vs. SMW 56
4.5 Discussion 57

5 Boosting Exploration with Stale Demonstrations **64**
5.1 Goal Conditioned Policies for Reinforcement Learning 65
5.2 Training a State and Goal-Sensitive Policy 66
5.3 Exploring with RRT 67
5.4 Experiments with Minigrid Environment 68
5.5 Results 70
5.6 Discussion 71

6 Scaling Up Self-improving Exploration **74**
6.1 Technical design of scaled-up turbocharging 78
6.2 Experimental testbed 82
6.3 Results 86

7 Conclusion **95**

Bibliography **97**

A Code Artifacts **113**

List of Figures

1.1 A deletion of a period in a configuration file did not crash the game, but rendered the videogame's progression meaningless. Compare (left character) the armor protection values from wearing underwear against the values when wearing endgame armor (right character). 2
1.2 A map of release dates of the Pokemon game franchise. One could interpret this chart as a developmental lineage tree with the latest games deriving core concepts and gameplay from the roots. Reproduced from Wikipedia [29] 4
1.3 A map of the major themes, projects, and overlaps of Ken's dissertation work. If the reader is ever lost in how my projects link together, refer to here. 5

2.1 A modified American Fuzzy Lop (`afl-fuzz`) fuzzer for playing games. Fuzzers are normally tools that explore software by providing different inputs, such as finding crash bugs. In this case the fuzzer is exploring software to find game winning inputs. (Reproduced from the IJON paper [4]) 18
2.2 Although this level seems impossible to beat, scripted testing ensures that the level is beatable. Any insanity seen is only added to give challenge and purpose to the player. Image reproduced from [30] 23

3.1 In Zelda, notice that the RRT algorithm fails to discover more tiles after a point due to its inability to discover the dungeon's door. As expected, Reveal-More touches significantly more tiles than a human game tester in the same time window. 40
3.2 In SMW, the algorithm performs better than the human player for several minutes. However, RRT begins to taper off in tiles touched while human exploration increases linearly. Reveal-More however trumps both aforementioned techniques from the start of execution. 40

3.3 Impact of varying amounts of algorithmic gameplay time versus tiles touched. As the amount of algorithmic play increases, the difference in tiles touched tapers off. 41
3.4 Impact of varying amounts of human input time versus tiles touched. The change in tiles touched as human input increases also diminishes when more human time is introduced. 42
3.5 Impact of varying the number of states extracted from a single 25 minute gameplay. Too few states does not give enough places for RRT to start, too many and it similar to measuring tiles touched from the human gameplay trace itself. 43

4.1 A static report resulting from applying Differentia to a game design change. The left side shows an overview of the moments reachable in the two versions of the game (color coded by version). The right offers filtered views of individual data clusters (zoom for details) including both average and representative sample screenshots from both versions of the game. 47
4.2 The first image (left) shows the local structure of a cluster (using the corresponding glyph from the overview image). The images to the right of the scatterplot visualizes (in order) a single sample screenshot from version 1, an averaged image of all version 1 screenshots, a sample screenshot from version 2, and an averaged image of all version 2 screenshots. . . . 49
4.3 SMW vs. GravHack: Game moments are divided into discrete clusters. Some clusters have strong biases towards one game version or the other, however the clusters with solely moments from one version suggests that this particular moment was present in only one version of this particular playthrough. 58
4.4 SMW deletion: In this cluster, a moment was present in SMW, but not GravHack. In GravHack, Mario flies off screen during this cutscene, causing the game to become unresponsive. 59
4.5 Most of the gameplay between SMW and GravHack remained the same. Visually, most levels did not change, however the changed user perception of progressing the level was not well captured. 59
4.6 Red vs. Brown: Major content changes show a major shift in moments. Strongly different visual content produces clusters that are placed far away from the other version. Whatever remains similar congregates in the middle. 60
4.7 New content: Some moments remained the same, such as the starting house for both Pokemon Red and Brown. Differing moments are illustrated in the bottom row, such as a new Charmander sprite for Pokemon Brown. 61

4.8 SMW vs. SMW (no changes): All clusters contain moments from both versions, and there are no clusters that contain moments from only one version. 62
4.9 SMW unchanged: In two human gameplay traces from the early game, there does not seem to be any difference in how the moments were grouped, indicating that Differentia considers these moments present in both (identical) versions. 63

5.1 Neural network architecture used to train an action policy. Masks represent a gate used to hide specific training data. 72
5.2 Two different map designs in the MiniGrid environment. Human demonstration data is made available only for Map A as if Map B had just been produced as an incremental game design change and had not yet been seen by human playtesters. Players move the oriented red triangle through rooms and doors to reach the green exit tile. 73
5.3 Efficiency of exploring the training (Map A) and testing (Map B) maps. When applied to a novel map, only the policy that was trained to consider both state and goal vectors reliably improves over the always-available RRT-Uniform benchmark. 73

6.1 Although One Finger Death Punch has far less inputs to the game that meaningfully advance gameplay, no one will argue that this game is less complex than that of Minigrid. 75
6.2 Reproduced from Zhan et al. *Taking the Scenic Route* [108], RRT exploration progress in Super Mario World is improved with the help of training from self-play data. 78
6.3 A single branch of an RRT tree is illustrated here. Each red node contains a vectorized RAM snapshot of the gameplay saved at that moment, and each segment connecting the nodes are represented by a button press combination. Images of gameplay are not stored, but are shown here to help understanding. 80
6.4 The Turbocharger NN learns how to predict a button press distribution given a current RAM vector and goal RAM vector. Doing so, we can imitate how a player moves around a playable space since we save branches of gameplay that contain starting and ending RAM states. 90
6.5 Previews of the 4 incremental versions of Super Mario World, as well as an unmodified Super Mario World level. From top to bottom, they are: Unmodified, Banzai 1, Banzai 2, Impossible, and Trivial. 91
6.6 Behavior cloning clearly dominates in exploration quality, showing that RRT was actually a hindrance to exploration. 92
6.7 In unmodified Super Mario World, an exploration agent that acts like a human player (but not replaying button presses) explores better than what a person did in far less time. 93

6.8 RRT with goal-aware action selection produces inferior exploration quality compared to cloning the player behavior at specific game states. Weaknesses in game state representation and action selection are becoming apparent. 94
6.9 When egregious game design changes are introduced, we see a wall in exploration hit by all exploration techniques. Note that player behavior cloning still out-performs all other exploration techniques. 94

List of Tables

Abstract

Analysis of Incremental Design Changes in Video Games with Automatic Exploration

by

Kenneth Chang

Videogames are software systems expressed in structure code and data, but it's far from obvious how changing one bit of code will impact the experience a human player might have with the game. Traditional playtesting and quality assurance testing remains the gold standard for ensuring a quality gaming experience, however there are considerable resource and morale requirements necessary to ensure that testing procedures will unveil problems in the user experience. Further, these testing traditions do not scale down to incremental, in-development, builds of videogame software, placing the entirety of videogame quality control at the ends of vast periods of coding.

With the advent of advanced AI gameplaying algorithms, I present a way to leverage gameplaying AI as an assistive tool for game developers. Rather than playing to win, these AI techniques aim to explore the playable areas of a videogame that a human player could encounter, potentially encountering areas of gameplay that developers did not intend to implement. I have developed two exploration techniques, one which relies on human gameplay traces, and one which self-improves with only a single seed of human gameplay. With this information, I also present a visualization workflow to generate visual reports of gameplay differences between versions of a videogame.

To my wife,

Oceane Bel,

whose invaluable support and love made this possible.

Chapter 1

Introduction

Everything about making a game is difficult [100], and occasionally what happens after release can be even more so. In December 2021, I finished the development of a *Rimworld* mod, and it was very well received by the modding community. At the time, I had approximately 30,000 active users of the mod, and in an effort to streamline the bug reporting process, I opened a dedicated forum thread to funnel all bug reports into a single place. Doing so allowed me to respond to each bug individually and mark them as patched or under investigation. My development cycle would proceed as such: a handful of bug-patching cycles, followed by either a minor or major content patch. Each content patch could expect several bug-patches to avoid software regression or to rectify oversights, and I grew to expect half a dozen (at most) posts per patch.

Imagine the shock I felt one morning when I opened my notifications to see nearly 50 new bug reports filed under last night's patch. Investigating further, all reports centered around a rather strange bug where an in-game character wearing any

undergarments would be completely invulnerable to melee weapons, bullets, fire, and explosives. Even more odd, the last patch pushed to players was a simple file-structure reorganization that didn't touch any code whatsoever. At least, that's what I thought it did. In reality, an errant period was deleted within an xml file that changed a single variable named `ArmorMaterialScaling` from 0.01 to 001. As illustrated in Figure 1.1, the game successfully ingested this '001' as a '1', making all undergarments provide a base 100% protection against all incoming damage.

Figure 1.1: A deletion of a period in a configuration file did not crash the game, but rendered the videogame's progression meaningless. Compare (left character) the armor protection values from wearing underwear against the values when wearing endgame armor (right character).

Frustratingly, while the game did not crash or throw an error, there was an immediate, perceivable problem in which the gameplay was delivered to the player. Suddenly, the entire point of the game's progression was thrown out the window: the best strategy was to wear a pair of briefs for complete invincibility. Fortunately, a fix for this problem was trivial, and players only had to endure this bug for half a day.

This highlights a problem in the game development process: **Code changes**

do not clearly describe the meaningful design/gameplay implications of those changes. Videogames are software systems expressed in structured code and data, yet it is also a delivery platform for experiences that have to be as mechanically sound as the software that delivers it. It's far from obvious how changing one bit of code will impact the experience human players in the audience might have with that game. If developers can know how gameplay changes even with the slightest code modifications, this knowledge can positively aid in the understanding of the game's developmental direction. This clarity can positively impact the experience in developing the game, and lowers the risk of shipping something that fundamentally breaks the experience of the game. I imagine a situation where game developers, and those invested in the process, have the ability to get some visual report on how the game evolved over time, with the ability to point at two discrete milestones in development and say "here are two pictures that illustrate key differences in the game." In a sense, having a game's developmental lineage illustrated in pictures and graphs, like shown in Figure 1.2 grants developers and investors the confidence to say that the game is going in the right direction.

A challenge to realizing this dream is the lack of players [105] and time [32] needed to test every incremental change introduced to the game. First, there will be no players available after each *incremental* code change to quickly play the build (and it would slow down development even if they were available). Hence, an automated solution is required. Second, the design wherewithal needed to interpret the impacts of a change is usually needed elsewhere [32], and likely game designers are already busy making the next incremental change. It would be onerous to have active developers

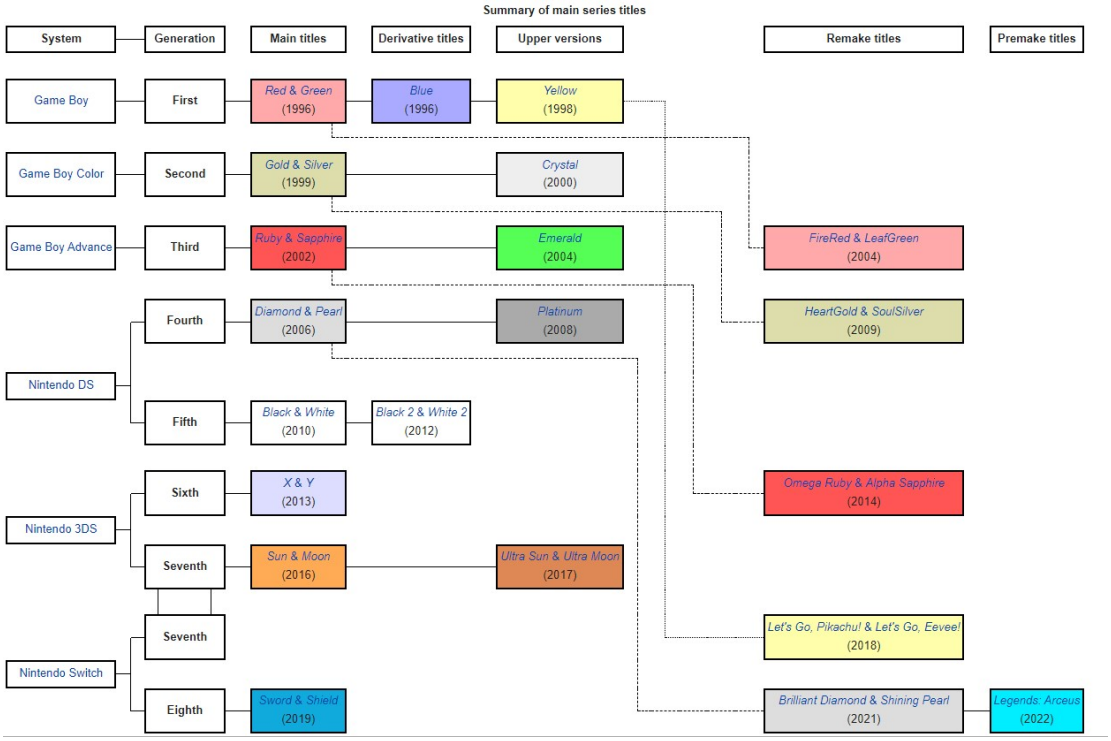


Figure 1.2: A map of release dates of the Pokemon game franchise. One could interpret this chart as a developmental lineage tree with the latest games deriving core concepts and gameplay from the roots. Reproduced from Wikipedia [29]

review hours of gameplay recordings even if they were available; there is for a simple report that shows them the space of play rather than a zillion samples of it. Finally, when humans are eventually involved in testing, the impacts of the expended effort should be maximally realised as easy-to-understand reports and meaningful data that can lead to actionable improvements.

This dissertation is concerned with technical methods for making those implications easy to understand quickly. The overlaps between my work, and the general themes of my dissertation, are illustrated in Figure 1.3. First, I implemented Reveal-More, a technique that amplifies the coverage gained from human testing. Achieving

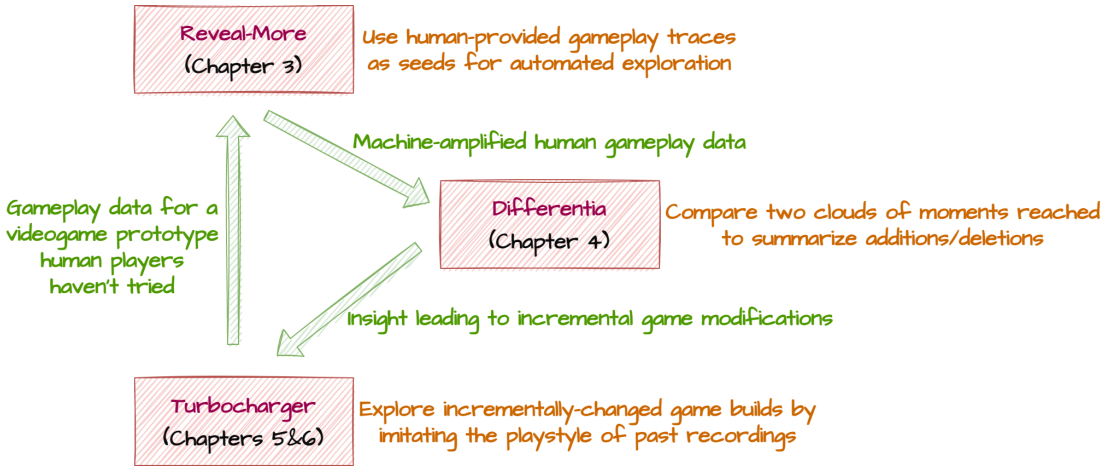


Figure 1.3: A map of the major themes, projects, and overlaps of Ken's dissertation work. If the reader is ever lost in how my projects link together, refer to here.

this, I can now extract a copious amount of gameplay data from a measured amount of human effort. With this gameplay data, I implemented the Differentia technique, which visualizes this extracted gameplay data across discrete versions of a videogame. Differentia achieves the visualization aspect of this dream. Finally, to move towards automation of this visualization, I implemented Turbocharger, which builds upon the RRT exploration algorithm used to gain amplified gameplay coverage. With this final component, I am able to move towards better exploration algorithms, and thus more automated gameplay exploration techniques to extract more gameplay data. I contribute the following three advances in computer science and game design:

Computer Science

1. Software engineering: Testing techniques tailored for testing game experiences.

2. Artificial Intelligence: Exploration techniques for gameplay discovery.

3. Information Visualization: Reifying noisy gameplay data into interpretable reports.

Game Design

1. Real Videogames: Software testing techniques are executed on culturally relevant games

2. Human Gameplay Data: Human player data was used to train machine learning models

3. Real Design Changes: Source code changes were applied to real games to create gameplay changes

Chapter 2

Background

To assist with the understanding of my dissertation, I have collected and condensed requisite information needed to understand my work. At a high level, my work focuses on advancements made in the topics of software engineering, videogame development, and automated gameplay using Artificial Intelligence (AI). This chapter primarily draws on resources outside of the academic sphere to position my dissertation work in relation to industry-accepted practices and goals. For topics specifically related to videogame development, much community knowledge is transmitted through sites like Gamasutra or in talks at industry conferences like the Game Developers Conference (GDC), etc. However, I will make special note of the few inroads in to this area that have been made by academics.

This dissertation uses specific terminology from the practice of developing videogames:

1. **Content**: Usually seen as orthogonal to the mechanics / logic / systems of the

game. Similar to the distinction between code and data. Content is seen by the player if the content is used by the game software to produce some outputs (e.g. graphics and sound) in a given play session. Example: art assets representing healing potions, a place in a virtual world where battle gameplay is done, etc.

2. **(Incremental) build**: An executable compiled from a specific version of software source code. Builds can be associated with a specific set of changes to the software of a game, allowing the quality of that change to be evaluated by interaction with the build. Incremental here implies minor changes to a build that ideally touches little beyond what was intended.

3. **Gameplay space**: The space of everything a player can do in the game. At the software level, this is relatable to the space of all plausible execution trajectories of the game software given specific, but different, inputs.

4. **Playability** [36]: Can a human player plausibly accomplish some gameplay objective? For example, can the player collect item X or complete level Y? Did the recent changes change that answer?

2.1 How game software development diverges from traditional software development

With so much time and effort spent on software testing practice and developing software test automation, it may seem obvious that these practices translate well

to videogame development. However, there are many key differences in how games are developed versus how traditional software is developed. At a surface level, one can immediately point out that the purpose of games is "fun" rather than strictly productivity [65], and this is correct. A survey conducted by Murphy-Hill *et al.* notes several key semantic differences among developer attitudes making games or office software [65]. That particular work also highlights several quotes taken from game developers, whose sentiment often describes the difference between game development and traditional development as a centralized focus on artistic expression, entertainment, and a shift away from rigid software correctness requirements (memory management, bug hunting, infrastructure correctness, etc.). The authors particularly note that because of this psychological difference, a need for videogame specific testing traditions and automation arises.

Another divergence from traditional software development is how game developers treat bugs and problems in the game development. While bugs in office software follow their own taxonomy [5], such as dependability and availability guarantees, videogame bugs follow a different taxonomy almost orthogonal to traditional software development. Lewis *et al.* documented one taxonomy of videogame failures [58]. These errors are a departure from normal software bugs (memory failures) and focus on gameplay bugs such as objects out of position, executing impossible actions, or players learning information out of order. The authors acknowledge the body of work done in creating taxonomies for normal software, and note that these bugs are an evolution of office software bugs, affecting complex gameplay rather than execution.

With Lewis' taxonomy, emerges a question: if we know what the bugs of a game are, can we use formal methods to prevent any of these events from happening? Since I am approaching game testing as getting to a game state where a failure occurs, it would seem that symbolic model checking [25] (automated logical analysis of software code) would be an ideal candidate for solving this problem. Surprisingly, there is already a body of work using invariant checkers [58] and technical specification generation [57] to ensure that a videogame cannot behave in an unintended manner. In these two related works, Super Mario World (a game that is extensively used in my dissertation here) is prevented from misbehaving though formal verification of what the game states can execute into, and any unwanted/invalid states are immediately bubbled up for developers to see. It would be as if this was the golden solution, yet implementation of these techniques has yet to see widespread adoption in industry.

I speculate that hesitation to adopt such practices in the game development industry is likely a psychological one. While the investigation of why this thinking pattern is far outside of the scope of this thesis, I would like to reproduce an excerpt from the conclusion derived in Murphy-Hill's work[65]:

> Overall, the results of the survey do confirm some differences. Based on statistically significant differences between Games and both Office and Other, we can say with some certainty that:
>
> - Game developers have less clear requirements than non-game developers.
> - Game developers tend to use what they perceive as an Agile process more than non-game developers.
> - Creativity is valued more in game development teams.

While it would be shortsighted to place the hesitation squarely on game de-

veloper's preference for creative liberty, I think it would be remiss not to indicate that there is resistance against any extra effort meant to thoroughly test games. Game design, for many developers, is an art that happens to utilize the medium of software. Hence developers treat bugs in games as *interruptions or challenges that have not been designed by the developers.* [55].

2.2 Academic Research on Software Analysis

An exciting solution to software testing is creating software that tests software, and this line of research has been thoroughly explored (primarily outside of games). Several key techniques have been proposed and developed in service of automated software testing.

Formal verification aims to prove specific properties about a *model* of software. Given a logical specification of a space of possible inputs, ensure that some logical predicate holds for all of the corresponding outputs [3]. Formal verification [48] aims to prove specific properties about a model of software (usually expressed using symbolic logic). These methods treat the execution of software as a series of logical constructs, and mathematically proves specific properties of the software's execution. If applied to videogames, we might want to check the invariant the player's level can only vary by at most one step at a time (e.g. they cannot progress directly from level 1 to level 3). **Symbolic execution** [62] generates formal models of code execution by analysis of the native program (assembled executable or source code) rather than re-expressing the

program with formal logic. In both of these techniques, the software in question can be tested to see if any invariants are violated. Applying formal methods to videogames, these techniques might tell the developer if the software behaves wrongly (level skips, ghosting through terrain, currency duplication, etc.), indicating that a problem exists in gameplay.

Yet another approach to software analysis is **test input generation**. This technique generates software inputs in a way that *explores* the state-space of the software in search for trajectories that demonstrate interesting behavior (e.g. crashing). The the Reveal-More technique 3 contributed in this dissertation is an example of test input generation. Specifically, my dissertation work uses Rapidly-Exploring Random Trees and behavior cloning whereas other methods from **software fuzzing** might be based on mutating example input seed [107]. For example, an Android software testing suite uses several fuzzers to provide the input needed to explore the software [18]. Most fuzzers treat the executable like a black box with predefined inputs, and searches the outputted execution states by changing the inputs in a deterministic manner [99].

While the promise of these software testing techniques may seem like a pre-existing solution to the problem motivated earlier, I believe that these approaches are not likely to see widespread adoption among videogame developers. As previously mentioned, many videogame designers and software engineers (who are likely also skilled software engineers) do not perceive themselves as traditional software engineers. Although they are well capable of using such tools, there is very little motivation for them to expend any effort on exotic testing techniques when artistic content remains to be

created.

A key distinction in my work is recognizing that the generated test inputs are not just blocks of data but are the result of interactive decision making that responds to what is on the screen and tries to achieve goals in the game's fictional world. While my work does fuzz the input space of games, it does by using algorithms that set and approach specific goals and make decisions that react to what's on the screen. I believe this reorientation of fuzzing is critical for making test input generation work appealing to practicing game developers.

2.3 Software testing

Traditionally, software testing has received attention from security researchers [83] and software engineers [9]. However, instead of focusing on usability, the goal of traditional software testing is to ensure that the program executes in the manner intended.

In games, however, execution correctness is not enough. Game developers are very concerned with whether the user experience is executed correctly, even if the software itself may not execute correctly. Thus the correct execution of code does not imply that the experience was executed correctly. For example, left clicking your mouse can execute code that shoots a gun, but if the gun inverts all of the colors on screen (and doesn't shoot), it is a failure in the execution of the experience. This does not mean that traditional software testing techniques are completely useless, rather I consider this as a

starting point from which I can fine tune its goals for experience testing. Instead of just quantifying performance and correctness, game testing tools should also consider likely player experience, and potentially prioritize experience over performance or correctness when considering game testing.

2.3.1 QA Testing

Traditionally, game developers expend a significant amount of effort to run Quality Assurance (QA) testing of their games, for both technical soundness and gameplay soundness. In QA testing, a frequent goal is to find examples of gameplay that demonstrate a problem to be fixed or to verify that a previously-known problem has been eliminated. The job of the game tester is often monotonous and repetitive, conducting regression testing, matrix testing, and functionality testing [90]. The tester's job is to make sure that any newly implemented features work as planned, and any bugs that were solved remain solved [35]. Towards this goal, QA testers are asked to continually revisit the game content, touching upon as many game paths as possible in every iteration of the game build. Overall, QA constitutes a major expenditure for game development organizations [54].

The resource requirement needed for proper testing is even more pronounced for independent studios [54], who spend a larger ratio of their funding on testing than AAA studios. Given this expense, software unit testing (which does not address gameplay issues) becomes the default testing framework for many developers. Software test automation techniques originally devised for other kinds of software are challenging to

apply to continuously changing design requirements, a commonality in game development [44]. The usage of such tools to comprehensively analyze source code does not necessarily translate to coverage of testing the qualitative features of software [1]. In cases where AI is used for testing, the turnover time to design and implement a new AI for a new videogame is significant [45], and outstrips the capabilities many independent studios can provide towards testing.

Because of the resource expenditure, game testers would rather have their time spent translate into more testing done. QA testing can require long hours where the testers are often fatigued from repetitive work and low pay [88]. Providing advanced tools for human game testers that amplify their testing efforts could lower fatigue, lead to more thorough testing, and open up new, high-skill career tracks for testers who can make best use of these tools.

If QA testing were to be eased for developers and testers alike, more testing could be done for every hour spent. Current research focuses much on removing the human aspect from QA testing as much as possible. Several implementations of automated QA testing have been proposed, and many are effective at testing specific aspects of a game. Xiao *et al.* [106] proposed an active learning solution to testing Electronic Arts soccer games for sweet spots. These sweet spots were the best locations a player could shoot the ball and maximize the number of points. By finding these sweet spots, Electronic Arts was given the opportunity to rebalance the game prior to release. Another project, ICARUS [82], used deep learning to learn how to detect bugs for the adventure game focused Visionaire Engine, completely removing the need for human-supervised

testing. In fields where game playing AIs such as those based on Monte Carlo tree search (MCTS) is used to play games as humans do [43], successful results have been observed for tracking down design problems in videogames. In further applications to playtesting [42], personas can be developed to figure out how players eventually learn how the game is played. In all of these applications, the sentiment towards testing involves removing the human as much as possible. However, humans are the only ones who can claim that a game is fun and/or engaging, and should be tasked to test those aspects of videogames rather than acting as human software drivers.

Nantes *et al.* suggests otherwise, that software agents can be used to supplement human QA testing [66]. My focus, in comparison, is on human game testers augmented by software to achieve greater game coverage in the same timespan. In the world of game development, it is difficult for AAA and independent studios to invest time and resources into developing better tools for QA. In all cases, the time needed to develop an algorithm that will play any build of a game can be arduous and slow, and with the rapid pace of game development the time expenditure needed is unfeasible. Hence, I believe that attempts at completely removing the human tester(s) from the QA cycle are headed in the wrong direction.

2.3.2 Continuous Integration

In software engineering, CI [31] refers to the process of automatically running build and testing tools in response to each change to the code or data of a project. It provides a method for the software development team to give assurance to users

and managers the assurance that development is headed in the right direction, at the cost of some flexibility and increased security measures [40]. While CI effectively automates the process of ensuring mechanical soundness of the software (crashes, memory leaks, flagrant bad behavior, etc.) it comes at measurable cost to a team's development speed [110] and could add creeping time overheads that no one wants to pay. Hence, the adoption of CI tests in game development will generate pain-points in development [103] that could lead to its eventual abandonment. Flatly put, CI in any software development is tricky to do well.

Further complicating the usage of CI for games is that automated tests do not necessarily reveal what experiences are actually experienced by the player. While CI systems can compare differences in code coverage [2] when testing, it is already clear that even code coverage fails to differentiate user experiences. Game developers are very concerned with whether the user experience is executed correctly, even if the software itself may not execute correctly. If code coverage is the only one analyzed, devs will fail to consider what is experiencable, and if the experience is the right one they wanted. [1]

Although it would seem that measuring all that is experiencable in a game requires one to play the game exhaustively, there are now automated exploration tools (further description can be found in Section 2.6.2)! As the name suggests, there is now software that can play games with the goal of exploring the game, not necessarily

[1] Grand Theft Auto 1 famously has a code bug that causes police NPC pathfinding to fail, causing them to recklessly drive towards the player, often killing the player. This is a flagrant code bug, but ultimately it saved the game from being cancelled because of the experience it brought to players. If a CI test caught this pathfinding code failure early on in development, and it was patched out as part of a normal bug ticket, this franchise would have likely never existed. (https://www.pcgamer.com/the-original-grand-theft-auto-was-almost-axed-saved-by-psycho-police-car-bug/)

beating the game. This is useful because if devs want to test a game, they want to cover all the ways to get to the end (or not!).

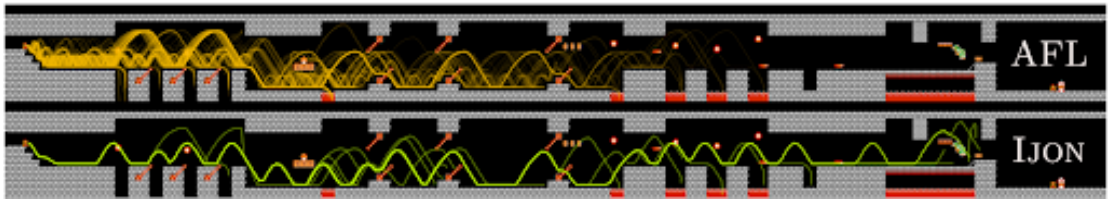


Figure 2.1: A modified American Fuzzy Lop (`afl-fuzz`) fuzzer for playing games. Fuzzers are normally tools that explore software by providing different inputs, such as finding crash bugs. In this case the fuzzer is exploring software to find game winning inputs. (Reproduced from the IJON paper [4])

To that degree, I want to remind the reader of the idea of software fuzzers, and how recently they have been put forth as a game testing tool. Fuzzers are tools that explore software to find corner case execution situations, and reproduce them for people to investigate. The goal of fuzzers is to use a scripted set of input to produce a desired result, and find the different avenues of input variations to produce that same result. For instance, if the goal of the fuzzer is to test a login feature, it may discover a workaround that does not require a username and password. As expected, a main use case of fuzzers is for security.

Wondrously, fuzzers can play games too. A modified American Fuzzy Lop (AFL) fuzzer [4] can play games deeply rather than widely (as in the case of standard AFL). Amazingly, this partifular AFL fuzzer can play culturally relevant games too! This demonstration of fuzzers playing games achieves the goal of beating games, but I want to consider this idea as a foundational concept of playing for coverage. Rather than strictly preventing or bootstrapping exploration around where the player is at any given

time, give the fuzzer more leniency to explore. What I see in Figure 2.1 are researchers applying traditional software testing techniques towards gameplay, with the possibility of further expansion into gameplay testing as well. Although in my research I will not use this particular method for exploration or game playing, the idea of exploring the area around the player permeates all of my projects.

2.4 Videogame development

2.4.1 Traditional testing of games

Game design is somewhat like chemistry: you are trying to produce a specific chemical reaction in the brains of your players. However, you cannot scan the brains of your players to verify that the specific reaction occurred. Further, if the reaction does not succeed, there is no undo button, and the results could be disastrous for your game studio. Casey DeWitt puts it plainly: "While no one is in danger of a collision in the world of videogames, there is the real danger of taking thousands of hours of developer efforts (not to mention perhaps millions of dollars) and releasing something substandard to the public, leading possibly to the closure of a studio and the endangering of many livelihoods. This danger necessitates a force that can act as its opposition – hence the combative nature, and hence Quality Control." [22] To do quality control, game developers enact playtesting schedules to test their game in progress against real players of the game. Their goal is to get instant feedback on the game, and to see if the players are reacting in the way the developers intended [33]. If not, changes are

suggested and brought to the drawing board, where they are addressed before the next playtest. The goal here is to make sure the game *feels* that it is going in the right direction, even if the software mechanics (e.g. bugs, unfinished art assets, etc.) may not be perfect.

However, playtesting should not be confused with QA testing. While both of these can help guide a videogame in the right direction, QA testing focuses mainly on the game's technical operation [87]. Quality Assurance in videogame development centers around testing the game for bugs and performance problems. This is often done with a combination of unit tests, dedicated testing teams, and more recently AI agents that interact with specific mechanics in the game to produce a desired result (described later in Section 2.4.2). Because of QA testing's more mechanical nature, many treat QA testing as separate from experience testing, and thus conduct testing more akin to human executed unit testing [35]. Regardless, the goal of QA testing is still to ensure that the game is moving in the right direction, albeit from the perspective of correct software execution rather than testing.

Although no one can argue that either type of testing (QA or playtesting) is individually superior to the other, it is becoming quite apparent that the costs of running tons of playtesting has ballooned considerably. In 2018, EA published work on using reinforcement learning based gameplaying agents to test games like a human does [10]. This work was rapidly followed up with academic work focusing on top-down shooters [111] and commercial research focusing on Hearthstone [34]. Because of this focus, I choose to focus this dissertation on maximizing the effectiveness of any human

testing done. If the costs to playtest are increasing, so should the benefit reaped from any effort expended.

2.4.2 Scripted testing of videogames

Before the rising costs of playtesting was identified, scripted testing of videogames was already put in place to address the rising cost of QA testing. In an ACM Queue article [19], the developers of *FIFA Soccer 1* (released in 2009) describe experiments with automation of the QA testing process. They created scripts that would programmatically provide inputs to the game over time to navigate between different screens of the game, verifying that they were still reachable. While the scripts could be re-run against different versions of the game, the scripts would often need to be modified as the layout of screens or the timing of interactions changed. In a GDC 2018 talk,[2] the developers of *Call of Duty* described how they integrated lightweight QA tools directly into their CI development flow. In response to each incremental code or data change, a tool directly launches the game into a set of specific scenarios in which it gathers numerical metrics (e.g. memory usage and frame rate) as well as reference screenshots for visual inspection. More recently, startups such as test.ai[3] and test.im,[4] companies outside of the traditional game development world, have advertised their automated QA systems as solutions to rising QA costs. Although QA is not necessarily a process to thoroughly test experience, it is an automated method that performs basic-level sanity

[2] `https://www.youtube.com/watch?v=8d0wzyiikXM`
[3] `https://test.ai/industries`
[4] `https://www.testim.io/`

tests to verify that the game's execution is not flagrantly incorrect.

Because of this well-defined goal, automated QA testing has matured the quickest in the realm of automated videogame testing. Unity, the developer of one of the most popular game engines worldwide, released a free automation framework for scripted testing of videogames.[5] This framework reflects how fast videogame design traditions are catching up with traditional software testing automation. Note how in 2004, tools such as Selenium emerged as a tool to execute interaction scripts on a webpage to conduct regression testing, and more recently Google Chrome added a record-replay feature to Chrome Dev Tools to record and replay a user interaction flow. And, as early as 2013, tools such as Timelapse [13] allowed one to "time-travel" as a feature to automate debugging by returning to states created during a human usage trace. This idea would be a precursor to my Reveal-More work, where I too use this time-travel mechanic to revisit places a human played through in a videogame.

While script-based testing can do a lot of things, and many of which are incredibly impactful to the end quality of a videogame, it really only assures the developers on a few specifics about the game being tested. Even if the script executes fine, and the route the script takes in a videogame executes as the developers intended, it does not imply that there is an alternate route of gameplay that the developers would hate for players to discover. In 2005, the developers of CloudBerry Kingdom [30] made an AI to create "insane" platformer levels. The "insanity" part of the level, seen in Figure 2.2, is generated to make the game feel interesting and challenging to the player even if it

[5] `https://blog.unity.com/games/on-demand-qa-testing-with-unity-automated-qa`

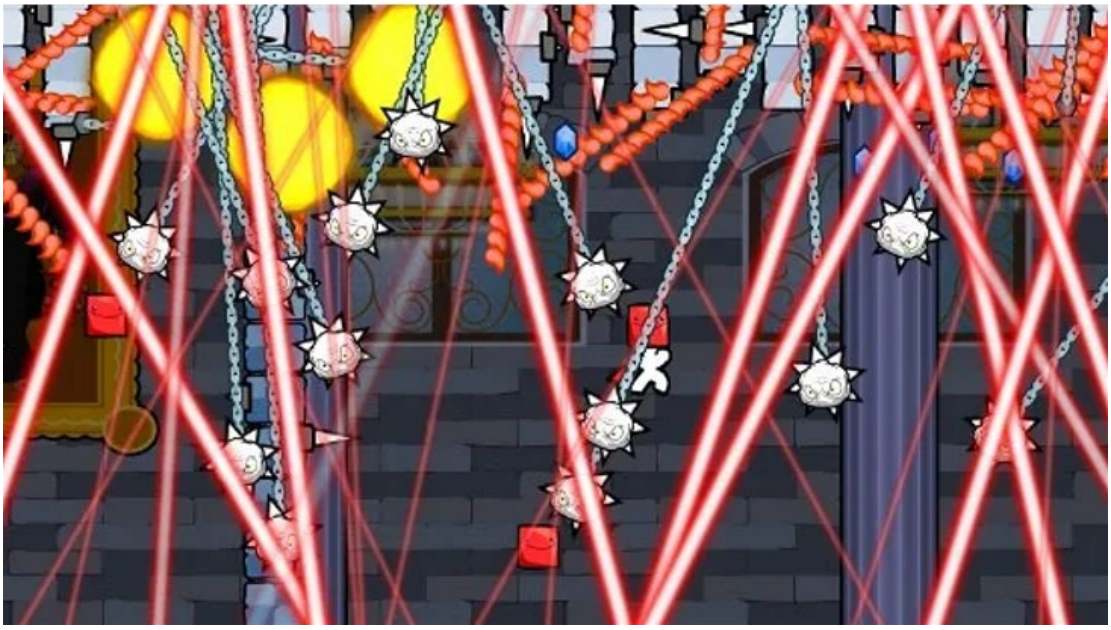


Figure 2.2: Although this level seems impossible to beat, scripted testing ensures that the level is beatable. Any insanity seen is only added to give challenge and purpose to the player. Image reproduced from [30]

may visually look impossible. But, to make the game playable, the AI is limited by the constraint that the level must be solvable, and it will play the game to ensure that it is, though this doesn't tell the developer if the level is solvable in multiple ways, and more problematically solvable trivially. Hence remains the need for human playtesting even if some testing can be scripted away.

2.5 Machine Playtesting

Machine playtesting, unlike automated QA testing, is a relatively newer term to describe the automation of playtesting, focusing instead on modeling the space of what players can and cannot do. In 2009, Smith *et al.* [95] introduced this term to

describe systems that are able to seek out examples of gameplay traces illustrating a specific property or proving that such a trace does not exist. This led to the development of Ludocore, where games to be analyzed are modeled in a logic programming language (with limited numerical computing support) in order for them to be analyzable. The tool could start the symbolic simulation in a certain initial state, and then ask for samples of trajectories that satisfies some condition (e.g. reaching a goal condition). In contrast to automated QA testing, the goal is not to tick off a checkmark (e.g. player health is never lower than 0 in this specific playtrace), but to report that there is/is not a way to achieve that goal. This work would later influence several strategies for asking and answering questions about a game in the absence of human players in "Game Metrics Without Players" [67], also by way of machine playtesting.

These efforts have the same goal as the work done in this thesis, however differ in one key aspect: they analyze and operate on *models* of videogames encoded in a restricted specification language. Even more recent papers that use exploratory AI algorithms base all experimental demonstration on models of games, such as using RRT as an intelligent level design assistance tool [7]. While no one will argue the relevance of applying these techniques to models of videogames, I believe that the impacts of any work I do in this thesis will be greater if I base all experiments on culturally relevant videogames.

In 2013, Smith proposed the open problem of designing reusable gameplay samplers that could answer realistic machine playtesting questions for realistic games [94]. The authors identified the WalkMonster system used in the ongoing development of

The Witness (later released in 2016) as an example of machine playtesting existing in the commercial world. This tradition continues in later systems such as *TesterTron 300* for *Thimbleweed Park* (2017). Both WalkMonster and TesterTron 3000 don't have a specific idea for what to test for (in contrast to automated QA), rather they explore the space of play to see what is possible and once finished, leave it to the human developer to interpret their behavior as intended or not (also in contrast to automated QA). This idea of exploring, finding what's possible, but leaving it to a human to interpret the report would later influence my work with Differentia, since the goal now is to make the interpretation better/faster.

For developers, scripted testing is a natural evolution from game-specific scripts created by individual developers to engine-level or tool-level support. Similarly, WalkMonster and TesterTron were successes on their scale, and Ludocore was a success at the engine level, but it could only express small abstractions of games. This path now seems to lead to engine level support for machine playtesting, which has been explored in both Monster Carlo 1 [49] and 2 [50]. Both of the Monster Carlos attempt to add machine playtesting at the level of game engines (specifically Unity) for application to real games. Monster Carlo 1 uses the MCTS algorithm to explore the space of play, using *restricted play* [46] analysis methods to answer design questions about the importance of specific game design changes. Monster Carlo 1 required some idea of a score or victory to guide the search process, however, in contrast to my work where a specific goal is not needed to create reports on game design changes. Monster Carlo 2 is an evolution of Monster Carlo 1, which upgraded the exploratory capabilities with deep

learning methods inspired by AlphaZero. These upgrades allow the system to transfer gameplay knowledge gained in one specific game state to similar game states. This idea of using neural networks to come up with feature vectors representing game states as well as transferring knowledge from stale human demonstrations would become the core idea for my Turbocharger projects, which too boost their exploratory capabilities from old data.

To close off this discussion on machine playtesting, I want to point to a very recent effort in industry to conduct automated playtesting using exploration on large scale 3D games. Researchers from the Google AI project developed game-playing agents that are tasked to learn how to play a game at a human level of competency [38, 53]. However, unlike related Google efforts to beat humans, this work achieves machine playtesting by fitting policies to human-provided demonstrations of play. The idea of fitting an action selection neural network to a seed of human gameplay has been explored as early as 2011 [85], but use of this idea as a playtesting tool is a recent development. The same authors would then leave Google to start Agentic, a startup that focuses on AI agents trained with human input data to playtest commercial games. As such, in my Turbocharger work, I believe that training agents to play like humans (not the same as replaying past actions!) can lead to excellent explorers of what a game has to offer.[6]

[6] I would like to thank Apex Legends (2019) for this particular discovery, whose matchmaker allowed me to connect with Chris Kirmse, who then became my boss at Athenascope, who then later connected me with his highschool friend, one of the key people behind Agentic.

2.6 Automated gameplay

To background the concept of automated gameplay, it is important to understand the continuing academic-industrial fascination with AI gameplaying. Although recent news would center deep learning as the key player in highly skilled AI for playing games, interest in a non-human playing competently at games has been a dream since the 19th century. The most infamous specimen of a machine playing at a near grandmaster level in chess could be attributed as early as 1770 with the Mechanical Turk.[7] Unfortunately (or fortunately) for the AI world, this magical machine was a sham that actually hid a real, human, chessplayer inside to act as its magical brain. Regardless of the implementation, the Mechanical Turk highlights the long-withstanding dream of beating humans at their own game without a human.

2.6.1 Playing games with AI agents to beat humans

The first instance of a real computer beating humans at a complex game can be attributed to Deep Blue's defeat of Gary Kasparav in Chess [68]. This success case from a mixture of advanced computing hardware and a tree-search algorithm called alpha-beta minimax. From that first success followed a lot of development in the use of Monte Carlo Tree Search for gameplaying, with followup success in games like Checkers and Connect 4. However, expert level play (optimial play notwithstanding) was never achieved with tree search alone.

The original DeepMind paper on Atari gameplay by Mnih *et al.* [64], inspired

[7] https://en.wikipedia.org/wiki/Mechanical_Turk

the use of deep learning (and neural networks) as a solution for beating expert humans at their own game. Rather than try to hand-craft a successful algorithm, researchers let a neural network self-teach itself how to play the game. Part of this self-teaching relied on a copious amount of human demonstration data. In particular, the famous AlphaGo [91], which beat Grandmaster Lee Sedol at Go in 2016 (then considered the ivory tower of AI gameplaying), relied heavily on human demonstration data to bootstrap self-learning. This technique evolved beyond just tree search, instead combining it with two policies of action selection and state evaluation, both implemented in the form of artificial neural networks. The ability to learn beyond-expert level play by training on traces of your own (less expert) gameplay is the heart of my Turbocharger work. In follow-up projects to AlphaGo, the AlphaGo Zero project [92] removed the need for human expert data, and surpassed the expert-level capabilities of its predecessor. This dramatic improvement in AI gameplaying has its costs however, and publicly we know that the hardware needed to achieve this cost $25 million alone. Given the limits of game development studios and testing automation budgets, I believe that approaches that use effective imitation learning techniques will be far more efficient.

2.6.2 Playing games with exploration-centric AI agents

In 2018, Uber research created a new algorithm called "Go-explore," [27] which was a gameplaying algorithm that was able to beat human records in Montezuma's Revenge. This particular feat stood out among the other gameplaying algorithms at the

time because Montezuma's Revenge favored exploration as a mechanism for increasing the game's score. In a way, getting to the "end" of the game (it didn't actually end until you died) required the player to skillfully explore every level until it maxed out points. Thus, its goal was not necessarily to find the shortest path to victory, but beat people in exploratory skills as a way to win. In the scope of my thesis, I too intend to beat people in exploration scores, however victory lies in the ability to cover more of what's doable in a game rather than maximizing a number.

Why is exploration the answer to finding the incremental game design changes? Exploring the game (rather than, say, diffing the code) will show us differences that are visible in the player experience while allowing us to look past technical implementation changes which end up not mattering. In "Taking the Scenic Route: Automated Exploration for Videogames," there are several exploration techniques demonstrated that attempt to maximize coverage of a videogame. In comparison to the exploration done in Go-explore, they tackle the problem of exploring a game's state space with the goal of producing a semantic map (useful for downstream inference tasks) on timescales comparable to human playtesting efforts. This map could give developers the ability to judge similar moments of gameplay, and allow inferences on how these moments relate to one another. This work also included many metrics to evaluate how well the exploration was coming along, an idea that I later use for all of my work, and is crucial to why exploration is a key solution to solving the problems posed in my thesis. Note that there are strategies that uses play-to-win algorithms in the service of exploration [8], however this strategy is not used in my work because my goal is to rely on some amount

of human data to bootstrap exploration.

Chapter 3

Exploring from a Backbone

The first pillar of my work is to amplify the testing coverage of a game given a fixed amount of human input. Attempting to maximize coverage of a game via only human gameplay is laborious and repetitive, introducing delays in the development process. Despite the importance of quality assurance (QA) testing, QA remains an underinvested area in the technical games research community. I show that relatively simple automatic exploration techniques can be used to multiplicatively amplify coverage of a game starting from human tester data. Instead of attempting to displace human QA efforts, I seek to grow the impact that a human tester can make. Experiments with two games for the Super Nintendo Entertainment System highlight the qualitative and quantitative differences between isolated human and machine play compared to my hybrid approach called Reveal-More. Reveal-More provides a powerful QA testing workflow that scales with the amount of human and machine time allocated to the effort.

3.1 Introduction

In quality assurance (QA) testing for videogames, conventional wisdom holds that automated approaches answer *software* questions (e.g. does processing this sequence of inputs yield the expected output?) and manual testing answers *gameplay* questions (e.g. will the game crash if I collect this item?). Nascent research efforts in automatic testing have tried to apply artificial intelligence (AI) methods to the problem of demonstrating interesting possibilities in play that developers might interpret to answer design and implementation questions that impact gameplay. So far, separated human and machine testing processes have shown complementary strengths [95], as expected [67]. Therefore, there is great potential in directly amplifying human tester effort to answer gameplay questions by using recordings of their play as the seeds for automated exploration.

Without automation, identifying inputs that lead to gameplay issues is a massive exploratory search problem that requires significant resource expenditure. Even in the simplest of videogames, there may be an astronomical number of distinct gameplay paths, only a few of which trigger a bug. In an ideal world, QA testers would indicate which span of a game is most relevant to them, and a system would quickly show them what was possible (or impossible) in that part of the game. Testers would save their efforts for directing, rather than enacting, repetitive gameplay experiments. Towards this goal, the foundational problem to solve is maximizing game state coverage in the service of encountering game design problems.

While there has been high profile successes in automatic gameplaying research [102], only recently has exploration specifically drawn attention [108]. Score optimization techniques such as Reinforcement Learning (RL) [64] and Monte-Carlo Tree Search (MCTS) [11] are setup to solve a different problem from the one faced in exploration. Techniques like MCTS may systematically avoid exploring certain play styles of interest simply because they earn lower scores. Additionally, the timescale on which automated gameplay techniques achieve useful results (i.e. minutes versus years of simulated gameplay) has only recently drawn attention [108]. For exploration to be useful in the QA process, useful reports need to be generated on timescales comparable to the pace of game design cycles (such as being able to provide feedback on weekly or daily game builds).

To demonstrate the usefulness of exploration in a QA process, I developed Reveal-More. This technique combines automatic exploration with just minutes of human gameplay, resulting in game state coverage that is superior to using each individual method alone. In such a manner, an automated method of exploration is used to amplify what a person can contribute to testing, thus lowering the strain placed upon testers to find all the paths in a game. To anchor the work in game development practice, I have carried out experiments in the commercial implementation of two culturally significant games. In several experiments with *Super Mario World* and *The Legend of Zelda*, there is up to a 5X increase in the quantitative exploration metric, and qualitatively illustrate the significance of increased coverage. Furthermore, this amplified coverage can be helpful in visualizing design changes and, in turn, help characterize the impact of

design changes.

3.2 Reveal-More Design

Reveal-More is a new technique for expanding coverage of the interaction space in a game starting from a sample of human tester gameplay. Reveal-More is aimed to be used by any team of game developers who do not have the time or knowledge to train a thorough AI testing agent for their game as long as adequate hooks are given to control the game remotely. The technique depends on an ability to save and load states at any point in a game. For this capability I used the OpenAI Retro Python library.[1] Retro is an extension of the OpenAI Gym library, which gives developers an interface to run automated gameplay algorithms on several different emulated platforms from the Atari 2600 to the Super Nintendo Entertainment System. Save states for the emulators are snapshots of the game platform's memory (and other stateful components) at any given time that allow us to jump between different moments in a game without re-playing the game from the initial states. The Retro library also gives us the ability to control these games programmatically (to provide button inputs and observe display pixels) and allows us access to key ranges of memory which enable game-specific tile count metrics.

The Reveal-More technique has two phases: data collection using human effort and amplification using automated exploration. In the initial step, a human game tester (in this work, one of the authors) plays the game as they normally would, making progress within the areas of the game on which a test is intended to focus. From

[1] `https://github.com/openai/retro`

this play, at prescribed intervals (typically a few seconds apart), Reveal-More records save states. These states are then used as the seeds for exploration. Algorithm 1 captures Reveal-More in pseudo-code, a function of human player data H, exploration granularity γ, the total exploration budget β, and an automatic exploration method E (parameterized by a starting state and an exploration budget).

Algorithm 1: Reveal-More algorithmic description

alg Reveal–More(H, γ, β, E)

Let $S \subset H$ with $|S| = \gamma$ be a selection of seeds

forall $s_i \in S$ **do**

  Let R_i be the result of running E from s_i for β/γ steps

  $R_i = E(s_i, \beta/\gamma)$

end

Output $\bigcup_i R_i$ all data seen in any exploration run

Once the game tester has finished their playtrace and the save states are created, Reveal-More takes over to amplify coverage of the game. The system starts by loading the first save state into the emulator and runs the exploration algorithm to cover more ground around that state. When the allocated time for the given state has elapsed (typically an amount of wall clock time equal to the intervals used in the first phase), the system loads the second state that was created, and algorithmic exploration restarts from that state. This process continues for all remaining save states from the first phase. In my experiments, I use the Rapidly-Exploring Random Trees (RRT) algorithm [56] as my exploration algorithm, which has been previously used to explore game spaces [108].

For the purpose of demonstrating amplification of human data, the details of the exploration algorithm are not particularly relevant beyond the fact that, given more time, the algorithm yields more coverage downstream from the given starting state. For action selection, actions are sampled from a weighted sample of the buttons players press when playing the game. The samples are generated from playtraces from the players, and determine the ratio of each button pressed from those traces.

In contrast to maximizing score, Reveal-More is designed to increase coverage. As both of the games picked involve predominantly spatial exploration, I operationalize coverage by *tiles touched*, which is a measurement of how many unique spatial tiles the player character occupied. The tile count metric is similar to the Walk Monster[2] implementation of exploration, where they are interested in the number of colliders touched instead of tiles traversed. My metric counts the level number the player has traversed through and the X and Y coordinates of the player, which is extracted from the game platform's memory. Every 16 pixels of horizontal and vertical position is counted as a distinct tile, and assigned a unique identifier based on the level and game mode. This approach is similar to other game-specific metrics such as *levels completed* in VGDL based projects [80], or the *number of rooms* explored in papers focusing in Montezuma's Revenge [27]. For games where the most significant aspects of game state are non-spatial (such as the inventory and character statistics for role-playing games), a different metric for coverage would be more appropriate.

[2] `https://caseymuratori.com/blog_0005`

3.3 Evaluation

To demonstrate the effectiveness of the Reveal-More technique, I applied it to two culturally significant games that have different notions of progression: *Super Mario World* (SMW) [72] as a 2D side-scroller action game and *The Legend Of Zelda: A Link to the Past* (Zelda) [73] as a top-down adventure game. ROM images for these games were obtained from public archives.[3] Both games sold millions of copies following their 1990s commercial releases. SMW is a linear game with few branching paths and a specific set of win conditions. Zelda, however, has many branching paths to the end of the game, and has non-linear methods to win the game.

I first wanted to know whether combining human gameplay with algorithmic gameplay can increase the area covered given comparable amounts of wall clock time, compared to exclusively human play and exclusively algorithmic play. I played the two games (in the same style one would for enjoyment purposes) for approximately fifteen minutes to create both my human exploration baseline and also the save states used in Reveal-More. In addition, several undergraduates[4] were also hired to provide their own, unbiased, gameplay traces of these games. In both games, a state is saved every three seconds. In the recordings, SMW gameplay corresponded to reaching level 2–2, Zelda gameplay demonstrated finding and finishing the first dungeon.

As a baseline for automated exploration, Reveal-More applies RRT to explore from each state saved in the human gameplay. This RRT implementation samples

[3] `https://archive.org/details/SNESRoms`

[4] I would like to thank Mitchell Pon, Nikhil Sheth, and Farhan Saeed for their gameplay contributions.

actions from a fixed game-specific action distribution for each game. RRT is started at the boot state of the game for SMW, and at the front of Hyrule Castle for Zelda. Instead of operating in the very high-dimensional space of visual pixel data, I apply two dimensionality reduction techniques suggested by Zhang *et al.* [109]. For SMW, I used the Pix2Mem embedding strategy to reduce the pixel observations into a 256 dimensional space. For Zelda I used a Principal Component Analysis of the first 8KiB of RAM into 64 dimensions (enough to account for 93% of the variance in the human gameplay recordings).

The action granularity (the number of frames for which a controller state is held before selecting a new action) for the two games differs since the gameplay is different. All automated gameplay holds each selected action for 6 frames in SMW, as it takes six frames for Mario to reach the maximum jump height, and 40 frames in Zelda due to the top down player perspective of the game. Given that there are no obvious directions to move (compared to always going right in SMW) having a coarser action granularity helped with exploration in Zelda. RRT exploration is run with these hyperparameters from its starting location with an amount of wall clock time equal to what was given to the human game tester.

I also choose to further experiment with the hyperparameters of Reveal-More, namely varying the amount of human input into Reveal-More, the amount of algorithm time allotted to a single human playtrace, and the number of states extracted from a human playtrace, all while holding the other two hyperparameters static. In one instance, I tested Reveal-More with different amounts of human input in increments of 5

minutes, saving 50 states per gameplay (with even divisions of time between each state), and exactly 15 minutes of algorithmic play. In the second instance, I vary the range of RRT gameplay from one minute to 25 minutes per state while keeping 5 minutes of human gameplay and 15 states. Finally, the last hyperparameter experiment extracted states every second from a long 25 minute gameplay, however I only use a proportion of the saved states in running Reveal-More. In this manner, I can demonstrate the effect of each hyperparameter on the exploration effectiveness of Reveal-More.

3.3.1 Quantitative Analysis

In Fig. 3.1 and Fig. 3.2, Reveal-More touches up to 2X more tiles in Zelda and 5X more tiles in SMW. The slope differences suggests a multiplicative increase in effectiveness in state coverage whenever Reveal-More is used. This proportional increase is not simply the result of running automated exploration, nor is it simply the result of summing independent contributions of two exploration strategies.

The sudden flatness of the scores in Zelda is explained by the time taken for the core gameplay to start. The RRT algorithm, started from the location where the player first encounters Hyrule Castle, was able to adequately cover the entirety of the castle grounds, however it fails to discover how to open the door to the first dungeon. The mechanism required for RRT to discover the entrance is to destroy a specific shrub on the top right corner of the castle map. Although the RRT algorithm destroys many shrub, it does not destroy the one required to allow entry after 17 minutes of gameplay.

When the hyperparameters of Reveal-More are varies, there are several effects

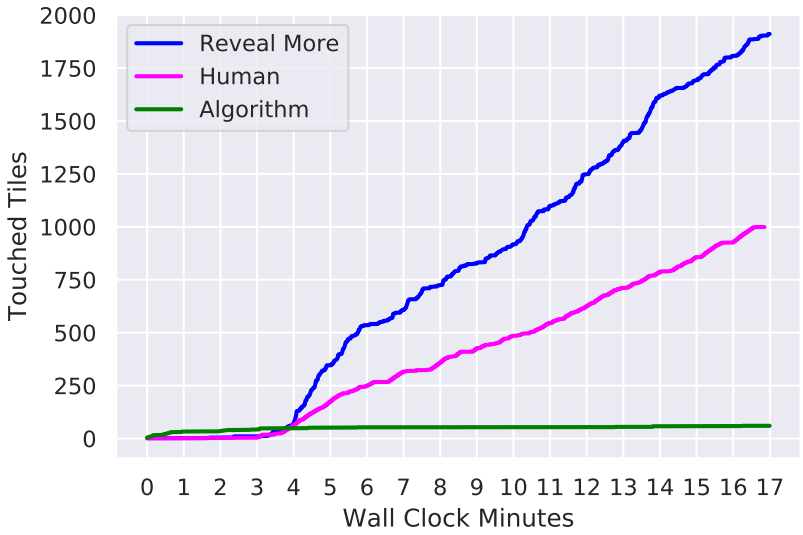


Figure 3.1: In Zelda, notice that the RRT algorithm fails to discover more tiles after a point due to its inability to discover the dungeon's door. As expected, Reveal-More touches significantly more tiles than a human game tester in the same time window.

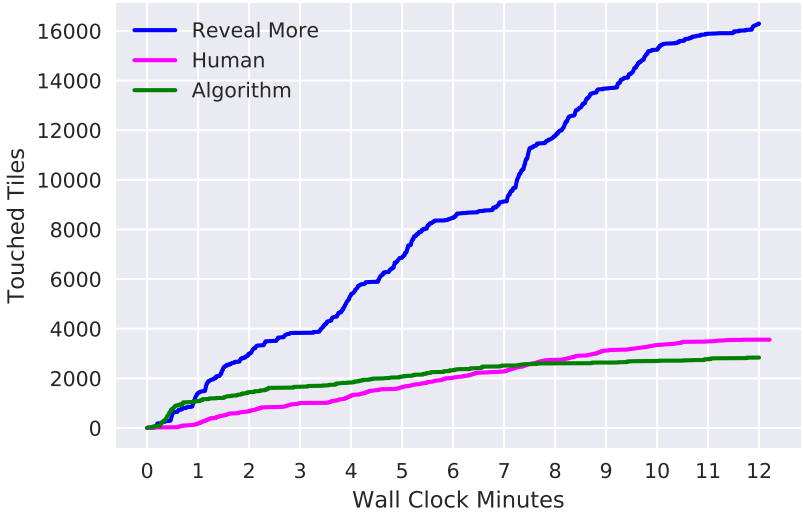


Figure 3.2: In SMW, the algorithm performs better than the human player for several minutes. However, RRT begins to taper off in tiles touched while human exploration increases linearly. Reveal-More however trumps both aforementioned techniques from the start of execution.

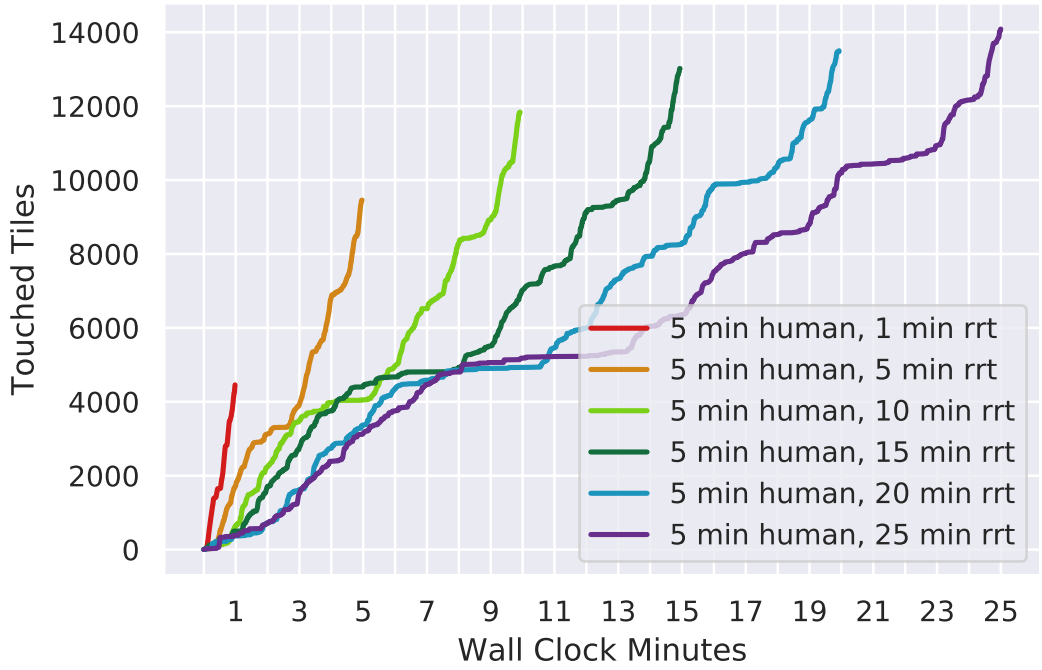


Figure 3.3: Impact of varying amounts of algorithmic gameplay time versus tiles touched. As the amount of algorithmic play increases, the difference in tiles touched tapers off.

on effectiveness of the technique. For clarity, only SMW is considered in these experiments. First, consider the impact of scaling the amount of automated exploration when holding the human gameplay under consideration (both the input dataset and the selection granularity) fixed. Results in Fig. 3.3 illustrate the same kind of sub-linear growth in final coverage as seen for the algorithm-only samples for Figs. 3.1,3.2 while growth within each run of Reveal-More is roughly linear. Next, holding the amount of exploration time fixed and increasing the amount of human data considered, I see similar trends in Fig. 3.4: roughly linear growth within a run and sub-linear response in final coverage. Together these show that either mode of exploration (human or algorithmic)

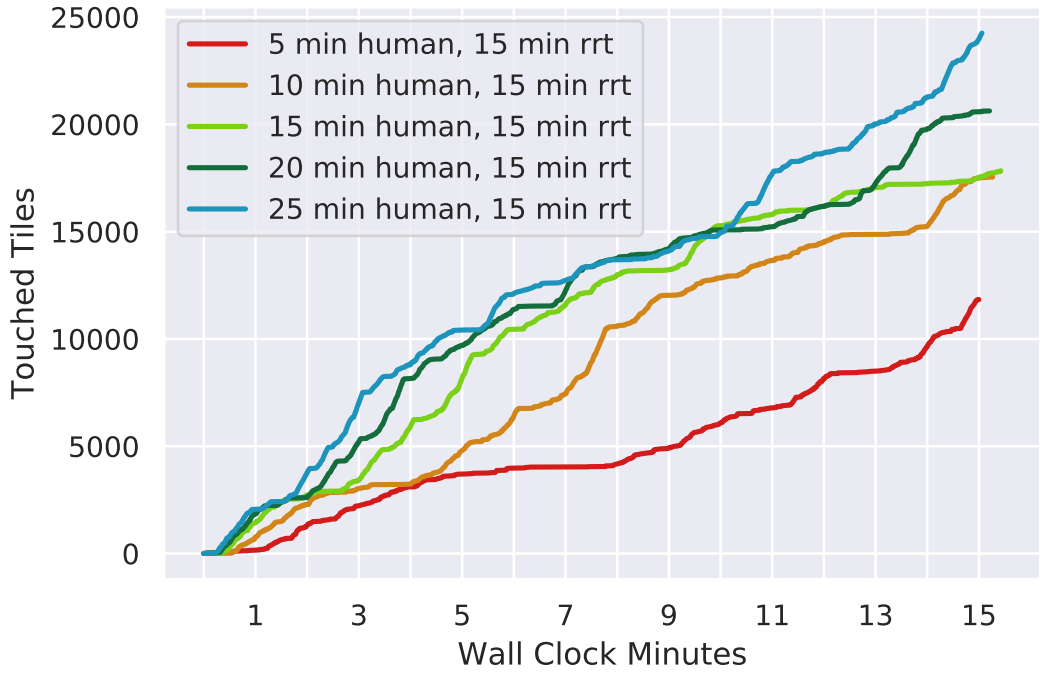


Figure 3.4: Impact of varying amounts of human input time versus tiles touched. The change in tiles touched as human input increases also diminishes when more human time is introduced.

faces diminishing returns when scaled in isolation.

To understand how the granularity of interleaving human and algorithmic exploration, I sweep the number of states considered as start points for exploration (holding the total automated exploration budget fixed). Fig. 3.5 shows that final coverage is maximized with a moderate number of starting points: using too many points with too little budget for the vicinity of each point to be sufficiently explored (or using too few points) invests exploration budget where the algorithm is facing diminished returns. The optimal balance surely depends on the game design, the diversity of play represented in the human data source, and the particular exploration algorithm. Holding all

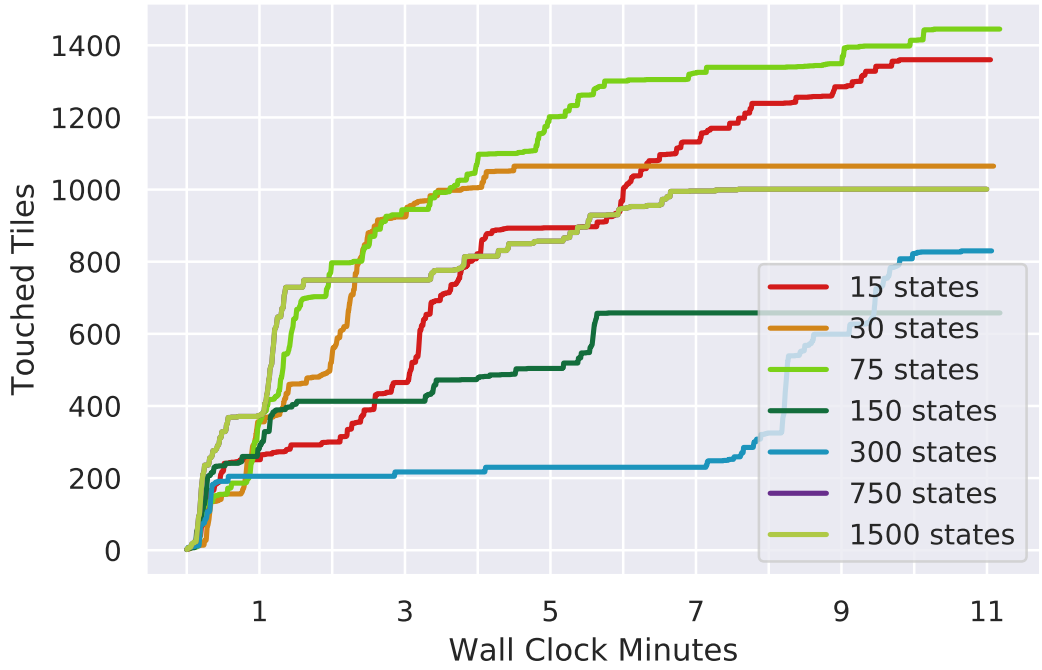


Figure 3.5: Impact of varying the number of states extracted from a single 25 minute gameplay. Too few states does not give enough places for RRT to start, too many and it similar to measuring tiles touched from the human gameplay trace itself.

other features fixed, I observe here that tuning the granularity of exploration results in at more than a doubling of the final coverage.

3.4 Discussion

The aforementioned results demonstrate the effectiveness of the Reveal-More technique on two culturally significant games, showing that Reveal-More is able to explore qualitatively and quantitatively more of the game's space with comparable wall-clock time spent. Using Reveal-More, a game studio can multiplicatively amplify the

efforts of QA testers to cover more moments in a videogame that could highlight problems or confirm fixes to them.

There is a weakness to Reveal-More in that it relies on a sufficiently intelligent exploration algorithm to explore around the backbone. Currently, Reveal-More either explores by pressing random buttons or from a fixed embedding of button presses learned from a previous gameplay. Therefore, final exploration quality is limited by how well the agent explores around the backbone. Later in this dissertation I will discuss a technique to improve exploration quality.

Further, simply gaining more gameplay coverage is not sufficient for testing purposes. I want to be able to visually differentiate design changes between different builds of a videogame in a simple to read report. Since Reveal-More is an algorithm that can find more moments in a videogame, the next logical step is to be able to show how two builds of a game have different potential moments in non-spatial videogames. This extension would be useful for developers and QA testers, because it would immediately show what differences lie in two builds. This current work already demonstrates some promise in differentiating accessible spaces in SMW, however I want to consider non-spatial changes such as increasing damage taken from environmental hazards or hitbox modifications.

Chapter 4

Visualizing Incremental Changes

Editing a single line of code in interactive software can have major impacts on the space of interactivity for the software. Minor edits can quickly shift a game from playable to unplayable, and the impacts of these edits are not readily understandable without thorough testing. Ideally, developers who implement incremental changes to software could directly examine the impacts on interactivity they have made from a visualization of the interactive space. As early as 1997, researchers proposed tools to visualize interactivity in Java games [98] and more recently have developed techniques that can visualize interactive stories [79]. In videogame development, there is an emphasis on verifying that the code created by a developer corresponds to the intent of what they wanted to create. There is a large gap between proper functioning of individual software modules and the overall player experience.

Because there is no oracle that can fully play and understand games, user researchers use human playtesters to encounter the significant moments of a game, re-

acting to them as normal players would. During testing, user research teams record these reactions, and compare them to the reactions the developers intended to elicit. Mismatches in reactions are returned back to the developers, who can apply the playtesters' feedback to the game to remedy problems in the game's playable space. While it would be nice to have a complete map of a game's significant moments, playtesting provides only a partial view of what is available in the game, bounded by time expenditure. Using Reveal More (discussed previously) a developer will be able to automatically gather evidence of the unique gameplay experience a game supports, but the question of how to summarize this space into a compact report remains. Hence, this chapter's work takes a step towards giving designers a tool to directly perceive the space that they are designing within.

Differentia contributes a way to visualize perceptual differences between incremental changes applied to videogames. The technique uses a (high-dimensional) gameplay map of a game's playable moments to build a two-dimensional map of differences between two successive builds of a videogame, isolating differences as visual reports that developers might quickly read and understand. This map is built by recording normal gameplay conducted by human players. I believe that this is a forward step towards improved development tools in continuous integration (CI) development environments, which emphasizes daily incremental improvements in a game's development cycle informed by automatically generated reports of software quality. I have operationalized Differentia in a software prototype and applied it to modifications of *Super Mario World* (for the Super Nintendo Entertainment System) and *Pokemon Red* (for Game Boy). To

demonstrate examples of reports Differentia creates, I will analyze the prototype visual representations generated and discuss its ability to convey design changes.

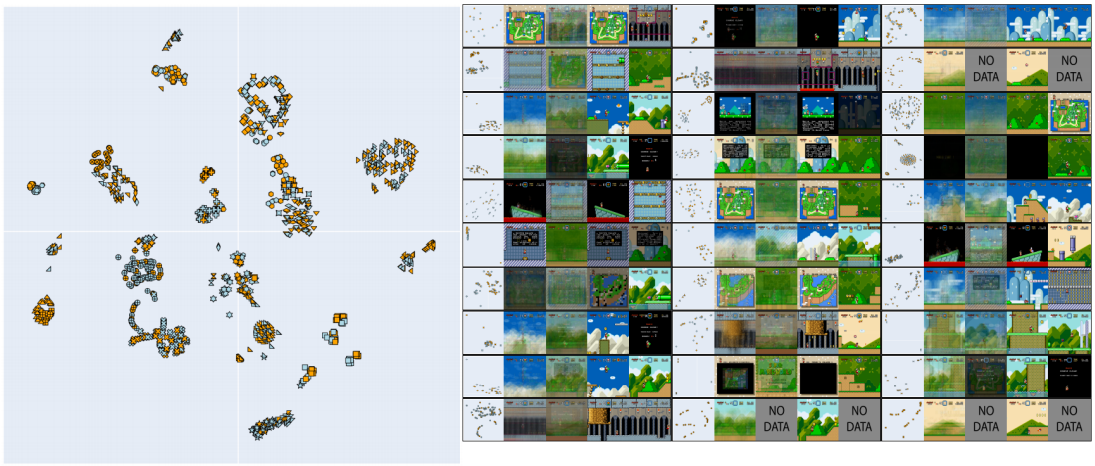


Figure 4.1: A static report resulting from applying Differentia to a game design change. The left side shows an overview of the moments reachable in the two versions of the game (color coded by version). The right offers filtered views of individual data clusters (zoom for details) including both average and representative sample screenshots from both versions of the game.

4.1 Visualizing Design Changes with Differentia

The goal of Differentia's visualizations is to provide an overview of differences in the experiential content present between two versions of a game, and from this overview provide specific details on demand for specific clusters of images. This zoom-in style of communication follows Schneiderman's mantra [89] "Overview first, zoom and filter, then details-on-demand," a generalized rule-of-thumb to help encapsulate and deliver information efficiently.

Figure 4.1 shows one of the generated reports from my visualization tool. It

shows differences introduced to the game *Super Mario World* by modifying the effect of gravity on the player character (more details on this experiment are given in the next section). The scatterplot on left side of the report offers an overview of the distribution of content seen before and after the design change. Each point in the scatter plot represents a single moment sampled from the gameplay data gathered from one version of the game or the other (again, details in a later section). Clusters of points indicate collections of similar moments. When a cluster consists of data mostly from one specific version of the game, it suggests those moments of gameplay experiences were only possible in one version of the game. That is, they were either removed or freshly introduced by the recent, incremental design change. The overall arrangement of clusters in the visualization, however, is not significant. It represents only a best effort to render the high-dimensional structure of game moments into a two-dimensional chart using modern dimensionality reduction techniques.

Detail views on the right examine one data cluster at a time. Of the five small images present in each row (Figure 4.2 shows two such rows, enlarged), the first shows a scatterplot of the local structure within the cluster, the next two images show the average appearance of all data in the cluster (grouped by which version of the game they came from), and the next two images sample point representative point (so that details missed in the averages can be examined). Because the grouping of data into clusters is automated in this technique (and thus not perfectly aligned with developer intuitions), the per-cluster visualizations can help discriminate significant from insignificant differences.

Figure 4.2: The first image (left) shows the local structure of a cluster (using the corresponding glyph from the overview image). The images to the right of the scatterplot visualizes (in order) a single sample screenshot from version 1, an averaged image of all version 1 screenshots, a sample screenshot from version 2, and an averaged image of all version 2 screenshots.

4.2 Technical Design of Differentia Prototype

To produce the aforementioned visualizations, I made a software prototype of the technique. Differentia is a collection of Python scripts that ingests a collection of screenshots generated from playing each version of the games being compared. From these screenshots, I train an autoencoder model that yields a high-dimensional vector representation of each image. The precise architecture of the autoencoder model is not significant here, and another technique that directly optimized the embedding of game moments to match their screenshots, e.g. a variational auto-decoder [41], would be a fair replacement. The high-dimensional moment vectors are projected onto the 2D plane using t-SNE [60], allowing us to render scatterplots. The high-dimensional vectors (not their 2D projections) are the clustered with K-means to yield (ideally)

interpretable groups for focused analysis. The specific implementation here is only one interpretation of the technique. One might consider alternative ways of representing game moments as vectors (see a comparison of alternatives by Zhan et al. [109]), other ways of projecting high-dimensional moment vectors into 2D for scatterplot display (e.g. UMAP [61]), or other methods of grouping comparable moments for inspection (e.g. spectral clustering [69]).

4.3 Visualization Design

The outputs of Differentia are: (1) an overview map of projected gameplay moments; (2) a collection of images representing the placement of gameplay moments for a specific cluster; (3) a collection of images representing the average image of each version's moments per cluster; (4) a collection of screenshots representing one frame of gameplay from a version per cluster.

Output 1 is the overview map so readers may see the entire space of mapped moments in one image. If they wish to read deeper into a cluster, outputs 2 and 3 provide information about each specific moment, showing the reader the mix of moments from each version as well as what that moment looks like on average. Output 4 gives the most specific detail, exactly one frame of gameplay, so a reader may quickly see exactly what was happening in this moment. These outputs are intended to roughly follow Schniederman's mantra: Overview first, zoom and filter, then details-on-demand (details accessed by zooming an achievable static image rather than interacting with a

complex interactive visualization).

Within the scatterplot, information is allocated to distinct visual communication channels. Because there are many bits of information in one graph, I have carefully chosen to allocate specific information to a dedicated channel of description. Specifically, the symbol used to map a moment represents each moment's cluster. Color represents which version of the game the moment came from, and the position of the symbol represents similarity to other symbols that moment is.

4.3.1 Gathering Gameplay Data

In my prototype, gameplay data is gathered by having a human [1] play each version of the game. There is no strict minimum time to play each version of the game, however gameplay should encompass as much content of the game as possible. Potentially, running algorithms that can explore the game or multiply human gameplay traces can be used to generate more images of gameplay to train the autoencoder. In the visualizations above, screenshots are taken every second of gameplay. The player was instructed to play each version of the game normally, and given a timer of 15 minutes to play each game. If the game failed due to crashes or bugs, the session would end there and any screenshots saved up until the crash are saved.

The Differentia prototype collects screenshots using the gym-retro Python library, which is an open source emulation and control system for retro videogames. This library allows us to screenshot the players as they play, and emulate the hardware that

[1] Lab members of DRL

runs the games. The games were obtained from public Internet archives.

4.3.2 Representing Game Moments as Vectors

To represent game moments as vectors, I trained an autoencoder on the library of images gathered from human gameplay. The autoencoder uses 2D convolutional neural nets to convolve images down into a 1D vector, referred to as a vector representation. These vectors represent the images in a compressed form, and normally an autoencoder can decode these vectors back into the original image. However, instead, I use these vectors and run K-means clustering to cluster the vectors into groups. K-means computes how close each vector is in distance, and groups vectors by closeness. The number of clusters depends on the game, because I consider each cluster as a unique moment of gameplay. For the visualizations, I assume that there are 30 clusters of unique gameplay, however this number can be tweaked to make the visualization appear better.

4.3.3 Grouping and Arranging Moments

Once the moments of gameplay are clustered, they are then rendered to images. Differentia uses Plotly [2] to make scatter plots of the game's moments. These will generate the images discussed in Visualization Design. Next, these images are placed into an HTML webpage. Although the images appear small in the window, the user is free to zoom into each cluster and view each cluster's details individually.

[2] https://plotly.com/python/

4.4 Example Applications

To demonstrate potential applications of Differentia, I applied it to modifications of commercial videogames, including changes to both static content and game mechanics. Specifically, I used two Super Nintendo Entertainment System games, *Super Mario World* and *GravHack*, and two Game Boy games, *Pokemon Red* and *Pokemon Brown*.

Super Mario World (SMW) is a platformer type game released for the Super Nintendo Entertainment System in 1990 while Pokemon Red is a role-playing game released for the Game Boy in 1996. I use the gym-retro [70] Python library to allow us to interface with the games, giving us the ability to run the modified games, inject controller events, screenshot gameplay, save memory snapshots, and return to gameplay moments saved in the past. This framework gives us the tools needed to capture gameplay in offline records.

While two of these games are commercial releases, two are modifications created by patching the commercial releases. GravHack is a modification of SMW where the effects of gravity upon Mario is halved, causing Mario to be able to jump higher and float down slower. The immediate effects of this change are easier jumping at the expense of being able to deftly control Mario. This modification to the game was made using Lunar Magic 3,[3] an open source ROM editor used by fans to create custom games within the SMW executable ROM. Pokemon Brown[4] is a fan-made content expansion

[3] `https://fusoya.eludevisibility.org/lm/program.html`
[4] `https://www.romhacking.net/hacks/134/`

of Pokemon Red, released in 2014. Although game mechanics remains the same, it adds new art assets to the game, as well as new regions to traverse and additional Pokemon to catch. This expansion was added to the game using ROM editing tools, and is emulated in the same manner as Pokemon Red. GravHack presents to us what a developer may implement during an incremental change in the CI workflow. Changing gravity in SMW is similar to a minor change in gameplay balancing, while Pokemon Brown can be seen as a major content release for a game in early access.

4.4.1 Incremental Change: SMW vs GravHack

In the overview image Figure 4.3, observe that distinct modes of gameplay have clustered into distinct groups. While many clusters are populated with moments from both game versions, some clusters contain data from just one version or the other: this is evidence that a numerical change to the game's mechanics has resulted in additions and removals of experiential game content.

A major standout is the cluster seen in Figure 4.4. This cluster detected that this end card scene was present in the unmodified version of SMW (right) and not in GravHack. During the gameplay of GravHack, this cutscene did not function correctly. Normally, Mario is supposed to jump onto a plunger to demolish the castle, however because the gravity was lowered, Mario jumps off screen and crashes the game. Hence, Differentia did not find any moments in GravHack that matched this moment in SMW.

For most other parts of the game, Differentia does not suggest obvious additions or removals. Because the way Differentia embeds game moments into high-

dimensional vectors in this prototype operates at the time between frames of animation (details in a later section), the most obvious change to the human player (floaty avatar controls) is not represented in the summary report because the player can navigate (floating or not) through almost all of the same game modes and levels. An alternate embedding strategy that was sensitive object velocities could separate floating moments into distinct clusters.

4.4.2 Extensive change: Pokemon Red vs. Brown

In a major content change, Differentia notices a very stark difference in the game's viewable content. In the space of the game itself, Pokemon Brown overhauls the entire game's art assets, including backgrounds, while leaving the layout and most of gameplay the same. One can see a formation where many of the clusters from one version are very far away from the clusters of the second. In the center, there is one big cluster of similar game moments. In the game itself, there are many gameplay moments that are the same, such as the menu layout, the RPG style top down perspective, and the battle screen. However, many of the art assets have changed, and because of that unique moments present in one game are visualized as absent in the other.

In the zoomed in cluster seen in Figure 4.7, there is one cluster in which a moment is present in both versions of the game (top row), and one moment present in only Pokemon Brown (bottom row). Overall, many clusters out of the 30 created with K-means were detected to only have moments from either Red or Brown, indicating that the games have very different gameplay moments. In reality, the content of Pokemon

Brown is extremely different: art assets have changed, backgrounds have changed, yet gameplay has not been modified. The bottom row of Figure 4.7 shows a specific moment in Pokemon Brown where this particular battle scene does not happen in Pokemon Red, demonstrating Differentia's ability to find visual differences. Despite major changes, it is significant that Differentia does not simply report that the entire space of interactivity has been added and removed as a whole—many structures can still be matched up before and after the change.

4.4.3 No change: SMW vs. SMW

In this experiment, I probe for false-positives in the change-visualization system prototype: does it highlight additions/removals when none have actually been made? Extracting two distinct sets of human gameplay data from a single version of the game, and then running the components of the Differentia prototype with different random seeds, it seems plausible that the system might identify and report some differences as an artifact of data collection and analysis despite their being no underlying game design change. At the same time, it would be encouraging to see the system abstract over these incidental differences and report the consistent structure of the unchanging game.

In Figure 4.8, there does not seem to be any clusters that predominantly contain moments from one (identical) version of the game or the other. The substructure within clusters is not identical (because, e.g. the human player played through levels in slightly different ways each time), but the system has nevertheless decided to place the

corresponding moments together as desired.

4.5 Discussion

The current implementation of the technique has two major limitations: it is not fully automated enough for CI workflows (it requires human gameplay for each game version analyzed), and human interpretation is needed to decide if a cluster that has been apparently added or removed represents a design problem. While the goal should not be to eliminate human intervention and interpretation, the next project should allow for human effort to be contributed asynchronously. Ideally, one should examine how a recording of gameplay in one version of a game can be automatically translated or adapted into an incrementally-changed version of a game. The ability to enhance automated exploration efficiency (of a fixed game) by pre-training on past records of human gameplay data has been previously demonstrated [108], but transfer has not been attempted across incremental design changes to the game. Therefore, a logical extension to visualization in a CI development process is to train gameplaying agents with a seed of human gameplay. Doing so, we can automate the exploration of a game and fit it into a CI pipeline.

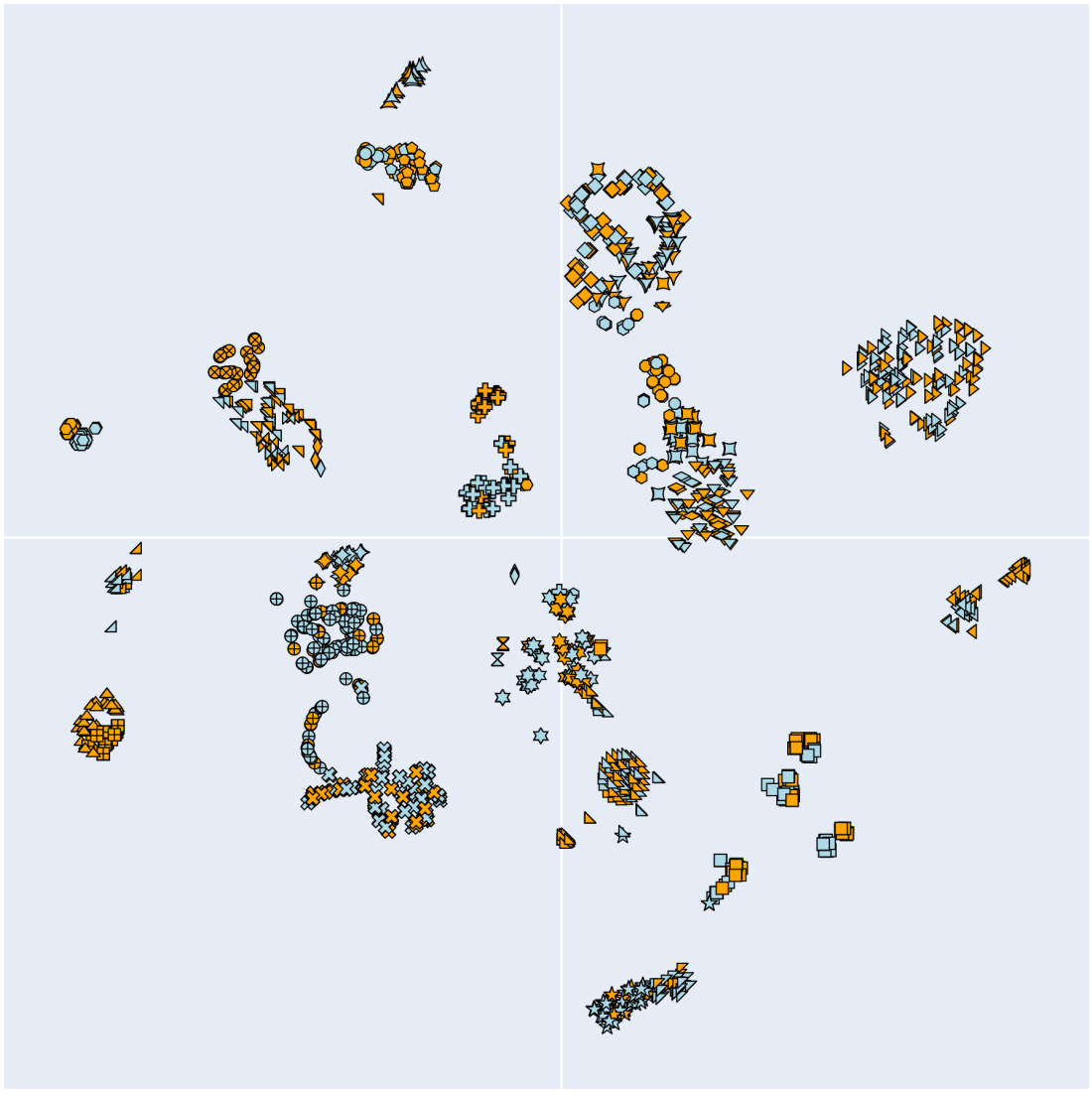


Figure 4.3: SMW vs. GravHack: Game moments are divided into discrete clusters. Some clusters have strong biases towards one game version or the other, however the clusters with solely moments from one version suggests that this particular moment was present in only one version of this particular playthrough.

Figure 4.4: SMW deletion: In this cluster, a moment was present in SMW, but not GravHack. In GravHack, Mario flies off screen during this cutscene, causing the game to become unresponsive.

Figure 4.5: Most of the gameplay between SMW and GravHack remained the same. Visually, most levels did not change, however the changed user perception of progressing the level was not well captured.

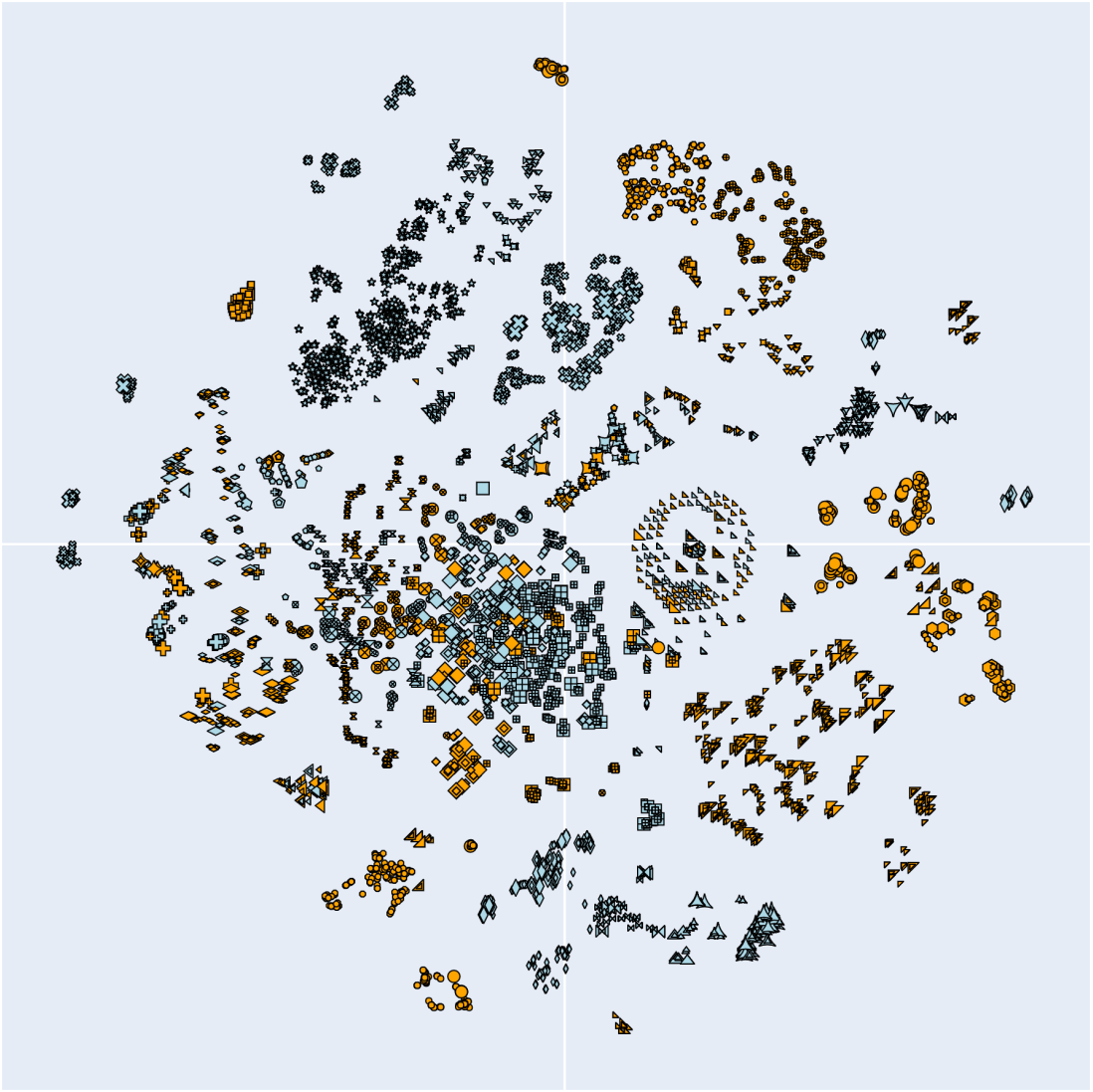


Figure 4.6: Red vs. Brown: Major content changes show a major shift in moments. Strongly different visual content produces clusters that are placed far away from the other version. Whatever remains similar congregates in the middle.

Figure 4.7: New content: Some moments remained the same, such as the starting house for both Pokemon Red and Brown. Differing moments are illustrated in the bottom row, such as a new Charmander sprite for Pokemon Brown.

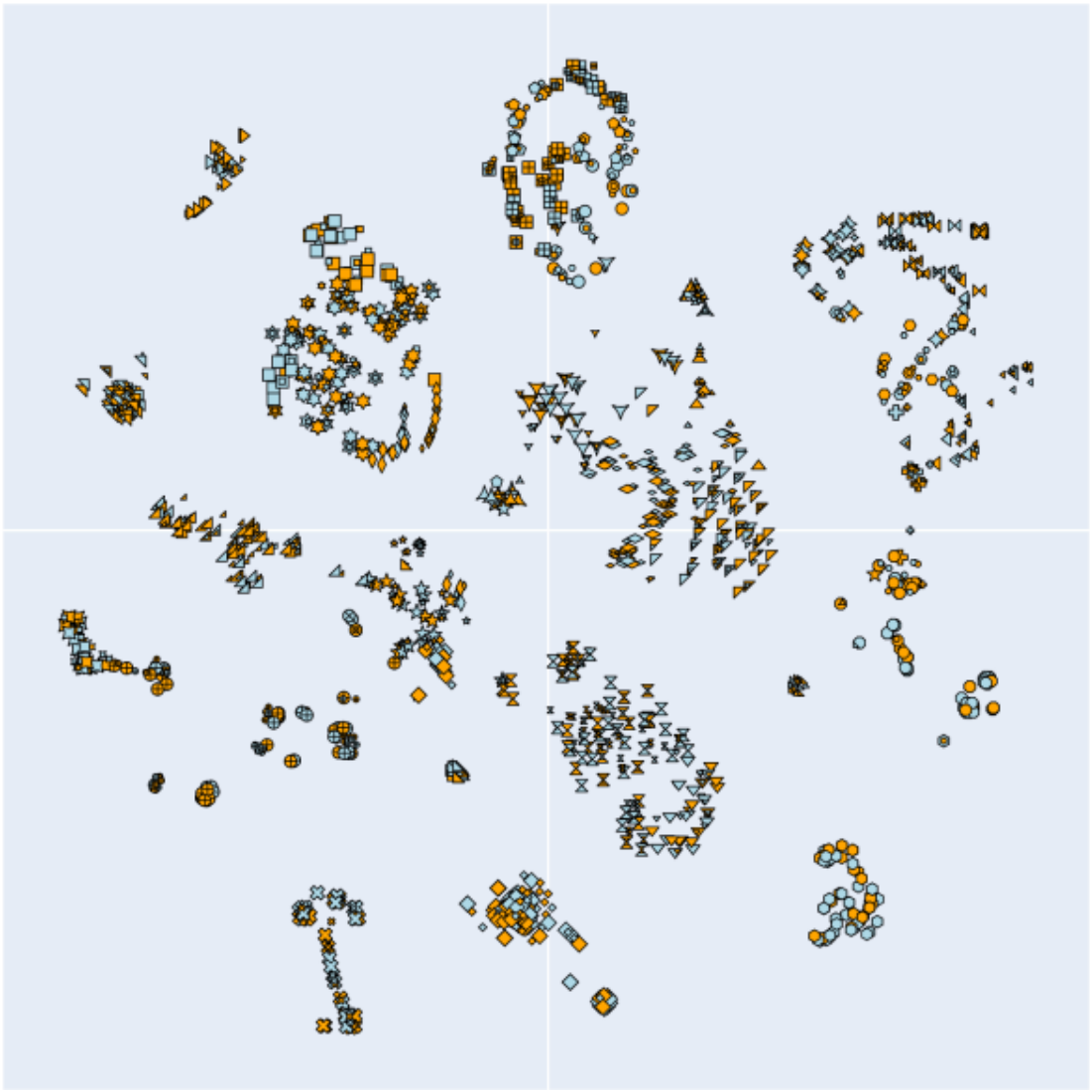


Figure 4.8: SMW vs. SMW (no changes): All clusters contain moments from both versions, and there are no clusters that contain moments from only one version.

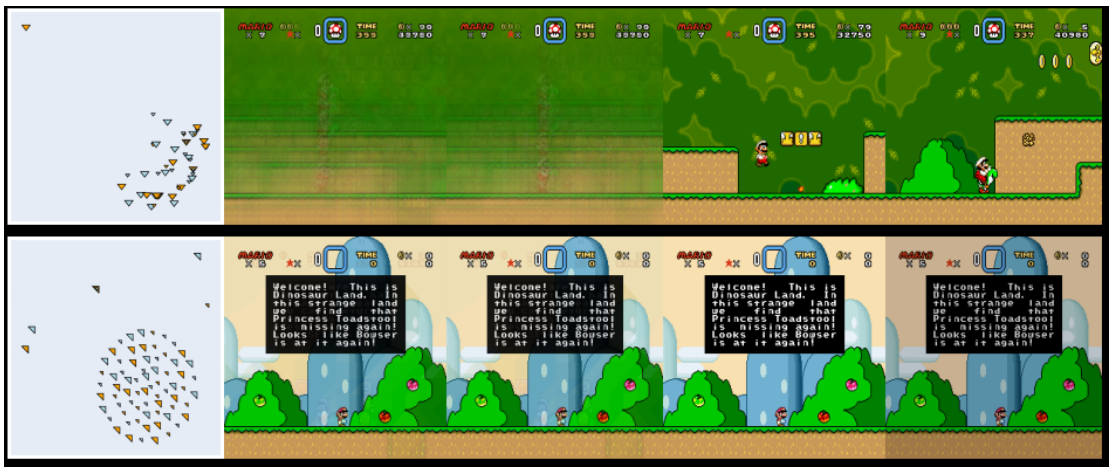


Figure 4.9: SMW unchanged: In two human gameplay traces from the early game, there does not seem to be any difference in how the moments were grouped, indicating that Differentia considers these moments present in both (identical) versions.

Chapter 5

Boosting Exploration with Stale Demonstrations

Automated testing is used in continuous integration (CI) pipelines for many major videogames [19, 101], however these tests usually do not attempt to map the playable space of games or report how that space has changed, which might reveal unexpected impacts of small code changes. To make this kind of machine playtesting usable within CI processes (where it will be applied to each incremental software change), it needs to be made efficient and *completely* automated. Reveal-More showed how to boost the efficiency of exploration using human player data, but this is exactly the kind of data that is not available after each software change. We believe that there is useful information to be extracted from stale data (e.g. from a human playtest of a recent build of the game), and that it could be applied to boost machine playtesting. In particular, we propose to use behavior cloning on the stale human data to train an

action-selection policy.

Reveal-More considers exploration algorithms based on rapidly-exploring random trees (RRT) [56], an algorithm from the robotics literature designed for exploring low-dimensional continuous state spaces that has since been applied to exploring videogame spaces [6]. However, RRT's exploration effectiveness relies on a key component: a subsystem for selecting actions that are intended to make progress towards a given goal state. The initial Reveal-More work uses very simple policies that were oblivious to the intended goal and even the current state. Naturally, the next step is to use machine learning to fit a goal-conditioned action policy to expert-relabeled human demonstration data (using another technique borrowed from robotics). I offer initial evidence of the usefulness of this approach in the MiniGrid [17] interaction environment, which has a low-dimensional state space with known bounds (similar to the setting for robotics applications).

5.1 Goal Conditioned Policies for Reinforcement Learning

The idea of learning to take actions in a way that depends on the agent's current state with explicit consideration for the agent's current goal has already been explored in the world of reinforcement learning (RL). Further, it has been shown how to distill unguided demonstration data into goal-conditioned action policies using a strategy called *expert relabeling* [23]. Without making any assumptions on the overall goal being pursued in the demonstration data, the relabeling technique defines a training

data set (pairing current and goal states with an action that makes progress towards the goal) by simply defining the goal to be some other state seen later in the trajectory.

In reinforcement learning research aimed at videogames (e.g. playing StarCraft or Atari games), it is common to evaluate policies on the very same environments used train them (in effect, training on the test data). As a result, trained policies can memorize a single brittle solution rather than capture robust playing styles [52]. In contrast, I require that this exploration policy accelerate exploration of a somewhat different environment from the one used to source the human demonstration training data. That way, one does not need to completely retrain an exploration agent to effectively explore a slightly changed videogame.

5.2 Training a State and Goal-Sensitive Policy

In the setting for machine playtesting for games, let's assume that a game dev has access to human demonstration data for a recent version of the game. This data is saved as pairs of game states and the human-selected action at that state. A typical human demonstration in our experiments involves 200 steps of gameplay in a turn-based interaction. This data is transformed into training data for a goal-conditioned action policy using expert relabeling. Concretely, for each point in the demonstration data, the goal is defined as whichever state was reached 1-20 steps further into the demonstration (sampling several random goal offsets). Considering expert relabeling as a form of data augmentation, this typically yields a dataset of about 6,000 examples.

For this early-stage work, the goal selection policy is a simple multi-layer perceptron (MLP) with three hidden layers, illustrated in Figure 5.1. The inputs to the network are the concatenated state and goal vectors (of three integer dimensions representing the agent's world position and orientation). The output of the network is a distribution over the few discrete actions available (up, down, left, right, and door-toggle). The networks typically have about 40,000 trainable parameters. The training of this model on the data described above using the Adam optimizer proceeds to convergence unremarkably. To create goal-oblivious or even state-oblivious ablations of this network, we mask (scale by zero) the input respective input vectors at both training and evaluation time.

5.3 Exploring with RRT

Exploration of a gamespace using RRT is done with the algorithm described in Algorithm 2. For the MiniGrid environment, I implement `sample_goal(configuration_space)` by sampling a grid location and agent orientation using a uniform distribution. I implement `policy(state, goal)` by asking the neural network for a distribution over actions, which is sampled to create a set of buttons to press. When moving from one version of a game to the next, the shape of the configuration space and the details of the initial state may change along with the details of the simulator. In the experiments `simulate(state, action` is always implemented by executing the same MiniGrid environment rules, just with a different level design. I imagine that for

many incremental changes, the inputs and outputs data types of the policy function do not change significantly (i.e. the game does not often add or remove spatial dimensions or actions).

Algorithm 2: rrt_explore(configuration_space, initial_state, simulate, policy, max_steps)

```
tree = new Tree(initial_state)

forall t in range(max_steps) do
    goal = sample_goal(configuration_space)
    state = tree.find_nearest(goal)
    action = policy(state, goal)
    result = simulate(state, action)
    tree.add_edge(state, action, result)
end

return tree
```

5.4 Experiments with Minigrid Environment

In this early-stage experiment with learning from stale human demonstration data, I focus on the MiniGrid interaction environment [17] (which has been used extensively in recent reinforcement learning research [84]). Figure 5.2 shows two examples of worlds seen in this environment. To model an incremental change to a game design, I sample two different procedurally generated MiniGrid maps and only provide human demonstration data for the first one. In particular, I recorded about 200 steps of nav-

igation between the various rooms of Map A, covering some but not all of the total reachable area of the map. The goal of this sampling is to use the human demonstration data from Map A to improve the efficiency of machine exploration of Map B. Note that the human demonstration data cannot be superficially transferred to the new map because (1) many paths in the first map are not possible in the new map because they would cross walls and (2) the demonstration touches very few tiles in the original map anyway. The choice of a simplified interaction environment and use of level generation to introduce variation is intended as a *computational caricature* [93] of incremental changes to navigation-oriented videogames.

We consider several versions of our RRT-based exploration agent:

- **RRT-Uniform**: Acts using a fixed uniform distribution.
- **RRT-Stateless**: Masking both the current and goal vectors, this model uses a fixed non-uniform distribution fit to the human demonstration data.
- **RRT-StateOnly**: Masking only the goal vector, this model can adapt its action distribution to the current state.
- **RRT-StatesAndGoals**: The full goal-conditioned action policy trained via imitation learning.

These variations were chosen to demonstrate the relative benefit of using stale human data to biasing the action distribution as well as the benefit of considering the current and goal states when doing so.

To measure exploration efficiency, I recorded the number of unique state vectors seen during an exploration run as a function of the number of environment interaction steps. Each exploration algorithm is run three times to reduce the effect of randomness from RRT. Better exploration algorithms could touch many more unique states without spending many steps revisiting old states. This metric for MiniGrid is comparable to the "tiles touched" metric used for more complex games like Mario and Zelda seen in previous chapters. I run each variation of RRT for 10,000 interaction steps and report the results in the following section.

5.5 Results

In Figure 5.3(a), the RRT agent trained with goal and current state data explores more unique tiles than RRT missing one or both or one of these inputs. As a result of overfitting to the human demonstration data that does not touch many states, the RRT-StateOnly model is unable to outperform RRT-Uniform (a policy that requires no access to human demonstration data). Although none of the exploration algorithms reached the maximum number of tiles that could have been explored within 10,000 steps,[1] it is clear that RRT boosted with goal and current state information can find unexplored areas in a shorter amount of time than the other methods.

When asked to generalize to a fresh map, Figure 5.3(b) indicates that both RRT-Stateless and RRT-StateOnly underperforms in the RRT-Uniform benchmark.

[1]Experimentally, I confirmed that all methods, including RRT-Uniform, were able to reach the maximum number of unique tiles when given enough time to search (e.g. 100,000 steps).

Only the combined model RRT-StatesAndGoals, which is trained to mimic the human's navigation style rather than their specific navigation path, is able to beat the benchmark. From this, I conclude that knowledge extracted from stale human data *can* be misleading, an appropriate model architecture and training regime can extract *transferable* knowledge that demonstrably improves exploration efficiency on incrementally changed game designs for which no human demonstrations are available.

5.6 Discussion

Where previous work trained a state embedding function from past gameplay data and used a fixed action selection policy, this work used fixed embedding and trained action selection policy. Under the assumption the game design changes incrementally over time, we believe both elements could be fruitfully learned simultaneously, even from collections of partially-stale data.

Naturally, the next steps are to scale up the Turbocharging idea of transferring exploration skill to more complex games. While the key concepts are simple, there are many questions left unanswered when scaling up. Namely, the representation of a game state, the representation of a goal state, and differentiating the states when measuring uniqueness.

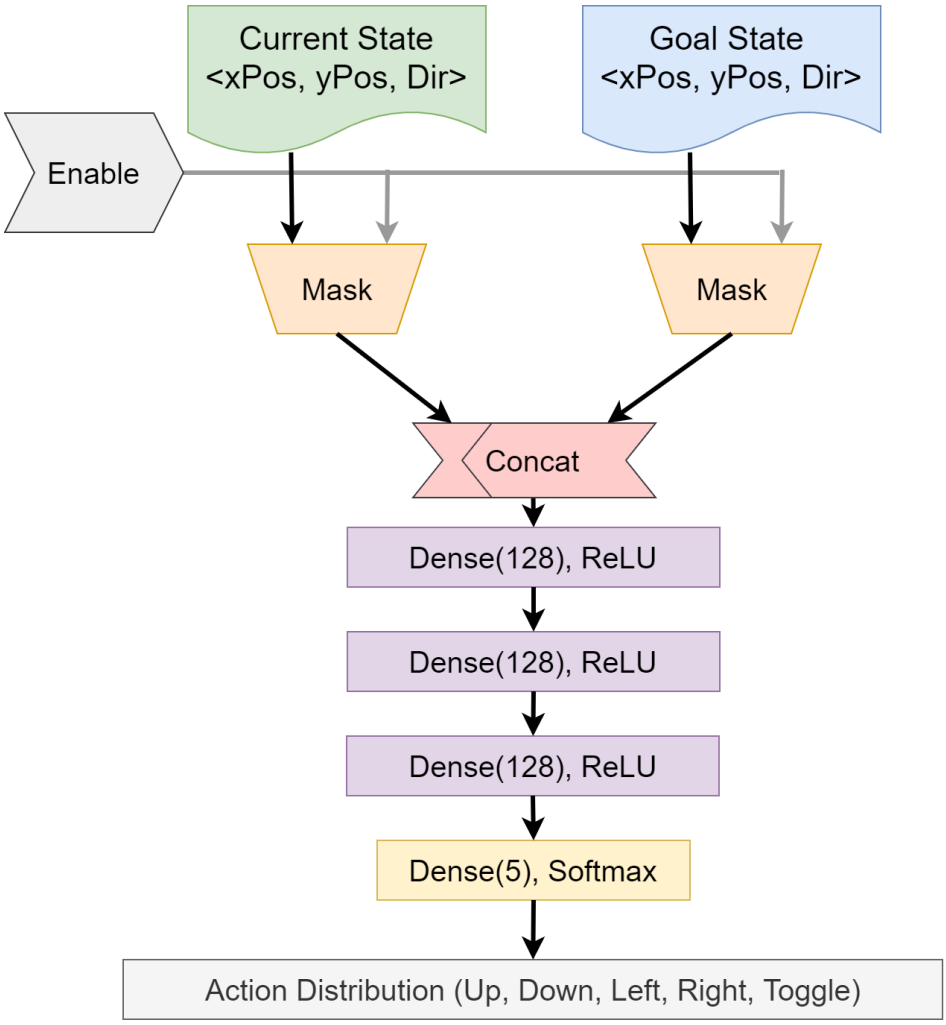
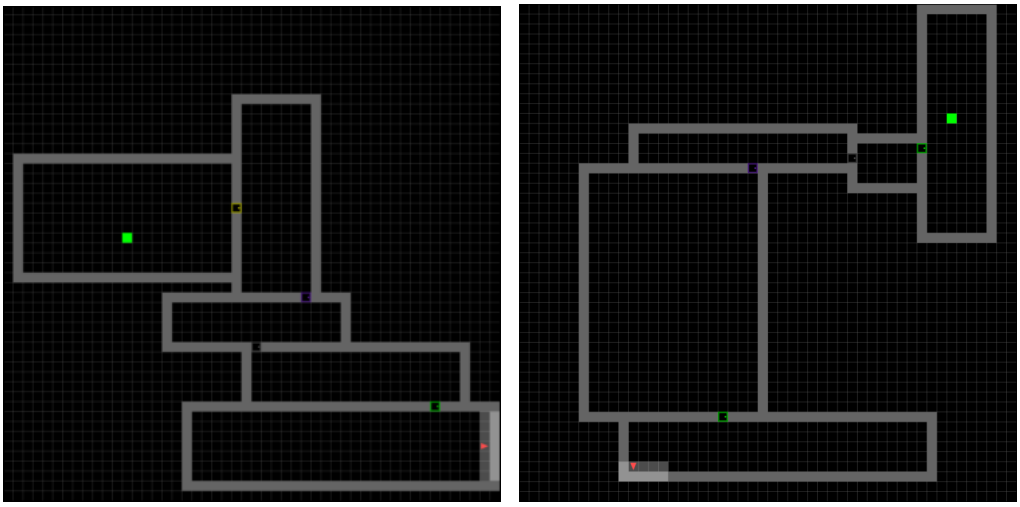
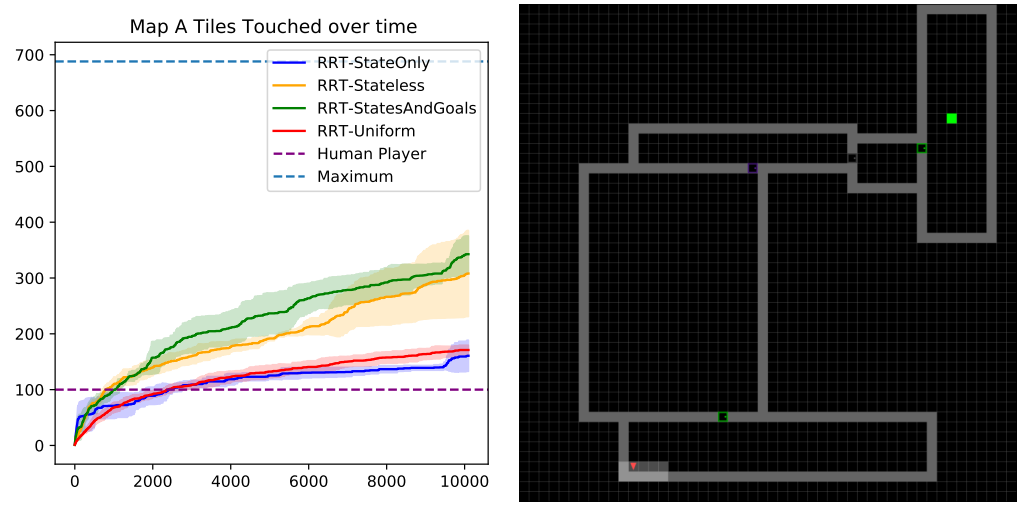


Figure 5.1: Neural network architecture used to train an action policy. Masks represent a gate used to hide specific training data.



(a) Figure A (b) Figure B

Figure 5.2: Two different map designs in the MiniGrid environment. Human demonstration data is made available only for Map A as if Map B had just been produced as an incremental game design change and had not yet been seen by human playtesters. Players move the oriented red triangle through rooms and doors to reach the green exit tile.



(a) Exploration of Map A (used for training) (b) Figure B

Figure 5.3: Efficiency of exploring the training (Map A) and testing (Map B) maps. When applied to a novel map, only the policy that was trained to consider both state and goal vectors reliably improves over the always-available RRT-Uniform benchmark.

Chapter 6

Scaling Up Self-improving Exploration

I have previously demonstrated how gameplay knowledge can be transferred across incrementally changed versions of a videogame. However, a key focus of my thesis work is the application of the proposed techniques on complex games. Although "complexity" in videogames is very much a philosophical debate, for the purposes of this work let us assume that there is a tangible difference of complexity between games. Further, this complexity is not a measure of the number of different input combinations. For instance, let's compare the games *One Finger Death Punch* (2014) and Minigrid: Although there is far more ways to input commands into minigrid to change the state of the game (up, down, left, right, toggle, etc.), One Finger Death Punch only has 2 buttons as input (left/right mousebutton). However, the latter game is exceedingly complex, with visual and auditory cues to guide the player's attention towards incoming enemies. Suddenly, two buttons feels like a tiny photo-lens in which a player can explore an incredibly detailed world of gameplay. An example of gameplay can be seen in

Figure 6.1[1].

Figure 6.1: Although One Finger Death Punch has far less inputs to the game that meaningfully advance gameplay, no one will argue that this game is less complex than that of Minigrid.

If I want to train an agent to learn how to explore complex games, I will need to improve several aspects of how I transfer gameplaying knowledge, and improve the ways I extract information from the game itself. Previously, I have also previously mentioned the work done by Google Researchers (whose researchers have now founded Agentic) in leveraging machine learning to create game-playing agents [38, 53]. Unlike the bombshell works of AlphaGo or AlphaStar, these AI agents are tasked to find out what is possible in a game by learning how to navigate the game in the manner of humans. A key departure from traditional gameplaying AI is the intentional focus on

[1]Gameplay image reproduced from `https://store.steampowered.com/app/264200/One_Finger_Death_Punch/`

game testing and automatic playtesting by folding in imitation learning inspired by the DAgger algorithm. Similar to the work I did in Minigrid, an expert plays the game on one incremental version of a videogame, and then the authors incrementally change the game to demonstrate transfer of exploration knowledge. However, there is a crucial difference between the two techniques. Consider this quote from the Google project's blog:

> Rather than ask developers to directly convert the game state into custom, low-level ML features (which would be too labor intensive) or attempting to learn from raw pixels (which would require too much data to train), our system provides developers with an idiomatic, game-developer friendly API that allows them to describe their game in terms of the essential state that a player observes and the semantic actions they can perform. [38]

I believe that while training from pixels may be too data intensive, there is significant information that can be gleaned from analyzing the memory data (e.g contents of RAM modules) of the game being executed (as seen in previous chapters 3 and 4). If I can get useful information from memory data (henceforth referred to as RAM), I can train an imitation learning agent that can navigate the videogame given a current RAM state and a goal RAM state. But, this brings up an issue of whether there is a natural/useful embedding of a game state into a lower-dimensional space where distances and angles there meaningfully represent gameplay differences in the actual state of the game. Mapping the game state to a lower dimensional vector has to meaningfully transfer the distance notion: without it the embedding does not clearly represent what the current state of gameplay is. However, this is the first of two challenges, the second is producing actions that are in the style of humans. In Minigrid, a single button press

meaningfully advances the state of the game. This is not so in Super Mario World, or for many complex games: fingers can press buttons in combinations so complex that simply pressing a few buttons for a number of frames is an oversimplification on how Super Mario World is played.

In the previous chapter, I did not have a term for this transfer of exploration knowledge between versions of a videogame. Now, I will call this "transfer of exploration knowledge between versions of a game using stale demonstration data" **turbocharging**. Turbocharging [59] in mechanical engines is the use of engine exhaust to improve engine performance. While I won't go in too much detail on how it works mechanically, at a high level an internal combustion engine requires a spinning compressor to compress fuel and air before ignition. The compression is normally powered by the ignition, however because the engine is likely driving something other than itself, there is a lot of energy lost in the shaft after ignition. To remedy the loss of energy, the exhaust gases from ignition is then fed into a turbine, of which the rotation generated is used to supplement the compression mechanism, thereby improving (turbocharging) the engine compression and thus gain more power out of the engine without extra energy (fuel) expenditure. This analogy is extremely close to the work I did with minigrid, and the work done in this chapter. In minigrid, I used the exhaust of previous gameplay to teach a neural network how to navigate a game, supplementing the learning normally done by an action selection neural network, thereby turbocharging the end performance of the neural neural network. Further, this *idea* of turbocharging has been worked on in the past on Super Mario World [108], and demonstrates improved RRT exploration progress

by training exploration with self-play data, as seen in Figure 6.2. In that situation, the turbocharging is done on stale data from previous RRT exploration, although in my situation I am turbocharging based upon exhaust player data. Therefore, it's quite apt that this work refers to turbocharging as the use of stale data in the service of boosting exploration.

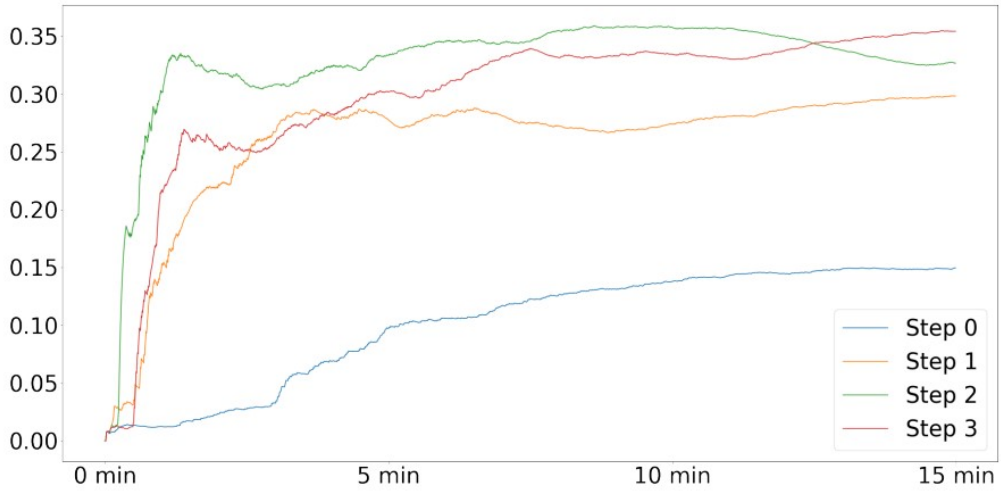


Figure 6.2: Reproduced from Zhan et al. *Taking the Scenic Route* [108], RRT exploration progress in Super Mario World is improved with the help of training from self-play data.

6.1 Technical design of scaled-up turbocharging

As a foundational goal for my prototype, I attempt to instrument the game in a way without source-code level changes. Note, the minigrid work required an embedding of the game state into a low-dimensional space where distances and directions in the embedding were meaningfully related to what happens in the actual game. To

do this, I vectorized portions of RAM dumps. In previous work done by lab members, Zhan [109] demonstrated that game states can be extracted by encoding RAM dumps from emulators into a lower dimensional space where distances in the projections represent distances in actual gameplay timestamps. These compact moment vectors are a representation of what's going on in the game, especially if one considers a small segment of the memory used in the videogame's runtime. In particular, Super Mario World's first 4KiB of RAM contains significant amounts of important data relating to the game's actual state, and this was done due to programming practices common in SNES development. Although this idea will likely not work in more modern games, it will be useful for testing turbocharging on a complex game: no source code instrumentation, yet a potentially useful window in understanding what the game is doing.

Like how Minigrid's exploration was done, exploration in Super Mario World is done with Rapidly-Exploring Random Trees (RRT). The rationale behind a tree for recording exploration is that each node in the tree represents a discrete movement in the playable space of Super Mario World. In Figure 6.3, I illustrate how a branch of exploration is saved. The left side represents the game state where gameplay started, as well as the current buttons held at that moment. Then, some number of emulated frames later, the gameplay arrives at the location on the right. In RRT exploration, the AI agent can choose to return to any point in this tree if it discovers that the goal it wants to move to is closer to a different point on the tree than it is at currently. This feature is important: being able to know how close a goal state is to where the agent already has explored means that any exploration done will meaningfully add to

the tree (and hopefully avoid re-treading on already explored areas). Each node in the tree also saves the first 4 KiB of RAM from SMW, milestoning the RAM at each point of gameplay. This tree data structure is also used during human gameplay to save gameplay traces.

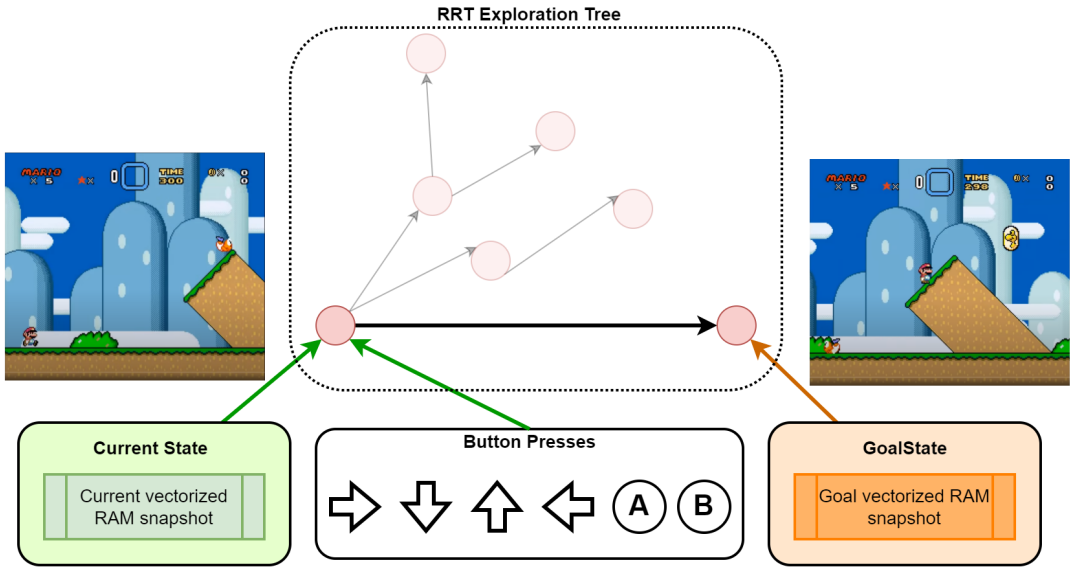


Figure 6.3: A single branch of an RRT tree is illustrated here. Each red node contains a vectorized RAM snapshot of the gameplay saved at that moment, and each segment connecting the nodes are represented by a button press combination. Images of gameplay are not stored, but are shown here to help understanding.

The exploration agent has also been overhauled to allow for goal-based exploration based upon memory vectors. Vectorizing the RAM is done with an autoencoder, seen in Figure 6.4. The autoencoder is trained to vectorize a 4 KiB RAM snapshot into a 128 dimensional array, and then unvectorizes it back to the same shape as the input RAM. This is similar to the Mem4KiB autoencoder model done in Zhan's work, and the one used in Differentia. Given a dataset of RAM, I was able to train an accurate autoencoder with a minimum of 90% reconstruction accuracy. Since RRT is used as

the primary method of exploration, differentiating memory spaces was implemented as measuring the binary difference between the two vectors. A higher binary difference indicates larger distance in gameplay.

Choosing an action is done with a separate neural network: `memmem2buttons`. In Reveal-More, a key weakness in RRT exploration was the action selection policy: it simply did not know how to move around intelligently in Super Mario World. This is not to say that it did not know how to explore: it turns out that randomly pressing buttons was a great way to explore around a backbone. However, there is no more backbone without a human playthrough, and this playthrough isn't available in a freshly changed videogame build. Therefore, memmem2buttons is tasked to mimic a person playing the game by learning how a person got to each node in a saved tree of gameplay. To do so, the network learns from a current vectorized RAM state and a goal vectorized RAM state to predict a button press combination. Previously, I discussed how a human playthrough saves RAM and buttons on every branch of the tree, and this is how this neural network learns to imitate humans: by learning how a person navigated between discrete gameplay moments, it learns how to imitate exploratory behavior demonstrated by the player. This imitative learning is never a mirror of what a person does because it can only predict the probability that a person pressed that button at that moment, however this inexactness makes the agent explore in the same style of a person: perfectly ideal for exploration. The AI does not need to mimic the player, rather it only mimics the playstyle.

Once memmem2buttons learns how to navigate a game space from a human

demonstration, executing it on a unseen version of a game requires a goal RAM vector. Without the ability to look forward in time, I created a goal RAM vector by mutating the current RAM vector. The mutation generates a new RAM vector with the same distribution of values as the current one. I concede that this is not a perfect representation of a future RAM state, however it is a better goal setting mechanism than that done in Reveal-More. Later, I will discuss the weaknesses of this approach, and if a goal state was ever needed.

6.2 Experimental testbed

To experimentally demonstrate the effects of scaling up, I created several experiments to demonstrate the transfer of gameplay knowledge between incrementally changed versions of Super Mario World. I hired a student[2] to create two incremental versions of Super Mario World, giving them the instruction to emulate a team developing Super Mario World. Because the original source code is no longer available, the student used an open source tool called Lunar Magic 3 to decompile the assembly of the game ROM and add their own changes to the game. This utility is very popular among ROM hackers, who pioneered the use of software like Lunar Magic 3 to create fan modifications of Super Mario World. Although this isn't true source code editing, it is the closest I can provide as source code modifications.

The experiments conducted use 4 incrementally changed versions of Super

[2]I would like to thank Wilson Mui for his contributions of alternative level designs, and all accompanying documentation.

Mario World (2 of which are not really incremental changes, but represent a critical error in development). The student was instructed to "add difficulty to the game" by increasing the number of enemies on screen. Previous of these levels can be found in Figure 6.5.

1. **Banzai 1:** In this version, there are additional Banzai Bills that spawn in the early stages of the first two levels. These enemies are large moving hazards that the player must jump around or else they die. From a gameplay perspective, this adds significant difficulty to each level.

2. **Banzai 2:** Similar to Banzai 1, there are more enemies to intensify the game's difficulty. However, Banzai 2 adds additional fast-flying Koopa enemies in addition to the large Banzai bills, requiring a different strategy to navigate the level.

3. **Impossible:** To demonstrate a potential error in incremental game development, this game version adds an invisible kill zone 50 tiles into each level. When the player hits this kill zone, they die immediately. Level completion thereby is impossible.

4. **Trivial:** On the opposite side of the spectrum, Trivial completely removes all enemies and pitfall killzones from the first 2 levels. Winning the level therefore becomes as easy as holding the right button for the entirety of the level.

Because the incrementally changed game versions are still fundamentally SNES game ROMs, I am able to emulate their execution in the gym retro. Therefore, I recycled

much of the instrumentation software used in Reveal-More 3 to extract information about the game. However, for the purposes of this project, emulating the game will cause retro to save the first 4K of RAM at every second of gameplay, as well as the buttons held at that moment. In addition, all this emulator data is saved in the RRT tree data structure as described in the previous section. Thus, if RRT is playing the game, any useful information about the game is saved as exploration is done. If a human is playing the game, the data is also saved in the same tree data structure, and the human is also able to reset back to the boot state to add more branches to the tree outside of the main gameplay path.

To get the training data used to train an action selection policy, I played an unmodified copy of Super Mario World for 15 minutes. I played the game normally, but reset the game three times during the playthrough to build three unique traces of exploration in the tree. I also focused my gameplay on the main levels of the first world, Yoshi's Island, since those levels were the ones incrementally modified. Once gameplay was saved, I trained the action selection policy and made it available for downstream experiments.

To demonstrate that the scaling up of exploration was actually happening, each game version is explored with 5 variations of the RRT exploration agent.

1. **RRT Aware:** This RRT agent behaves as proposed in the technical description section. The exploration agent is tasked to move to designated points in the RAM memory space for anywhere between 600 to 1200 frames. Effectively, this gives the player a random amount of time to go somewhere, and add its exploration

progress to the RRT Tree.

2. **Uniform Random:** As a control, this agent does not do anything other than press buttons uniformly randomly. The button to pause the game and to shut down the emulator is disabled to allow for reasonable execution.

3. **RRT Oblivious:** Same action selection policy as (1), however the "goal" vector is the same as the starting vector. This policy does not know where to go, and should ideally perform less well than RRT Aware

4. **RRT Random:** The original RRT for exploring SMW as proposed in Taking the Scenic Route. Uniformly random buttons are chosen as a way to get to a "goal" state.

5. **Behavior Cloning:** Instead of using RRT, this algorithm samples only the current state, and asks the action selection model what the player pressed at that state. It is similar to RRT Oblivious, but the exploration agent cannot reset back to previously explored states to build a RRT tree.

To measure exploration effectiveness, I will be using the tiles touched metric as described in chapters 3 and 5. All experiments are allowed to execute for 80,000 emulator steps, with each branch having a randomized play length of 600 to 1200 frames. Ideally, experiments should demonstrate that RRT Aware has the best exploration quality.

6.3 Results

After running the experiments proposed in Section 6.2, I graphed out the unique tiles touched over the 100k emulator steps. To reduce the effects of randomness inherent in RRT exploration, each exploration strategy is run 5 times with a different emulator random seed, and any deviations in the number of tiles touched are represented by colored shadows around the line representing tiles touched over time.

At a high level, behavior cloning surpassed all expectations and dominated in exploration effectiveness, as seen in Figure 6.6. This is not the result I was hoping for: it appears that the use of RRT is actually a hindrance to exploration quality. I will discuss potential reasons why later in this section, but for now it would be ideal to take a look at exploration quality over time for each game and RRT variation.

In Figure 6.7, we see that behavior cloning surpasses the exploration quality of all RRT exploration algorithms, and also surpasses the tiles touched by the player. This is an expected result, as effectively I created an expert that learned from imitating the player how to play SMW. Even though RRT is supposed to incrementally build a map of explored tiles, there is a lot of overlapping exploration done by resetting states, and without this handicap the behavior cloning agent was able to explore more of the game in the style of a human player (even if what the agent explored wasn't where the player went). Given more time, the agent would have likely played outside of the areas that the human player explored and thus unique tiles touched would have plateaued.

In both incrementally changed versions of the game, seen in Figures 6.8(a)

and 6.8(b), the RRT algorithms that use behavior cloning in any degree do not outperform simple behavior cloning. Unfortunately, this result demonstrates that RRT is having difficulty exploring the game even though it was trained on human demonstration data. However, it is evident that the behavior cloning model is a good explorer, and we are seeing the effects of that in its superiority of any of the RRT exploration algorithms.

There are several explanations for this result. First, even though the memory autoencoder was able to encode and decode memory into vectors, this is only a useful result for understanding where the player is at now. The proposed way to create a goal vector (mutating the vectorized memory state with the same distribution as the current memory vector), evidently is not a good way to select a goal state. As stated earlier, a key goal of scaling up turbocharging was to remove the need for source code access, and yet here we have evidence that removing source code access has removed the ability to know what a reasonable future memory state could be, since there is no way to explicitly create a future memory state. If I had source code access, I could write a mechanism that extracts precisely where Mario is at any given time, and correctly modify key gameplay information such that it correctly represents a goal state.

A second weakness discovered is the granularity of actions executed in the emulator: they are not representative of the buttons pressed by a person. In Minigrid, it is very clear that pressing a single key (e.g. the right arrow) will move the player character one square to the right. There is no ambiguity in that movement. In SMW, however, there is great ambiguity in methods of moving Mario. A person has near

infinite combinations of holding down buttons for any amount of time, and still get the same result of moving 5 squares to the right. Complicating this action selection are the advanced gameplay mechanics behind moving: holding down any combination of buttons changes how Mario moves even though the end result is the same, and further how Mario moves is a key gameplay mechanic that players can experience (spin jumping, normal jump, short hop, dismount hop, etc.). Because of the gameplay complexity in movement, simply predicting a button press combination without having extremely fine control on how long to hold each button compromises how effective the executed actions are at navigating the game space.

A third, and final, identified weakness in scaling up is the representation of the game state, and finding differences in the game state from RAM snapshots. A key assumption in the usage of vectors as game state representations is that vectors that have closeness in binary difference are actually close in experienced game content. This is unfortunately an under-estimation of closeness because there are many other mechanical parts of RAM that control unseen mechanics that change every frame, but do not actually change the gameplay experience. These mechanics are counters, sound buffers, color palette buffers, and other things that govern mechanical operations but not gameplay operations. Therefore, even though a single frame of gameplay has advanced, the changes in the vector may be so great that any differences detected in vectorized RAM is spurious.

Figures 6.9(a) and 6.9(b) demonstrate that in games where a critical gameplay flaw was unintentionally introduced, there are significant changes in how effective

exploration was for all algorithms. Any RRT algorithm plateaued at 100 tiles, and even the effective Behavior Cloning failed to explore as well as it did for any other game version. Although RRT + behavior cloning failed to explore well, this graph can be used to potentially determine critical gameplay problems introduced in an incremental change.

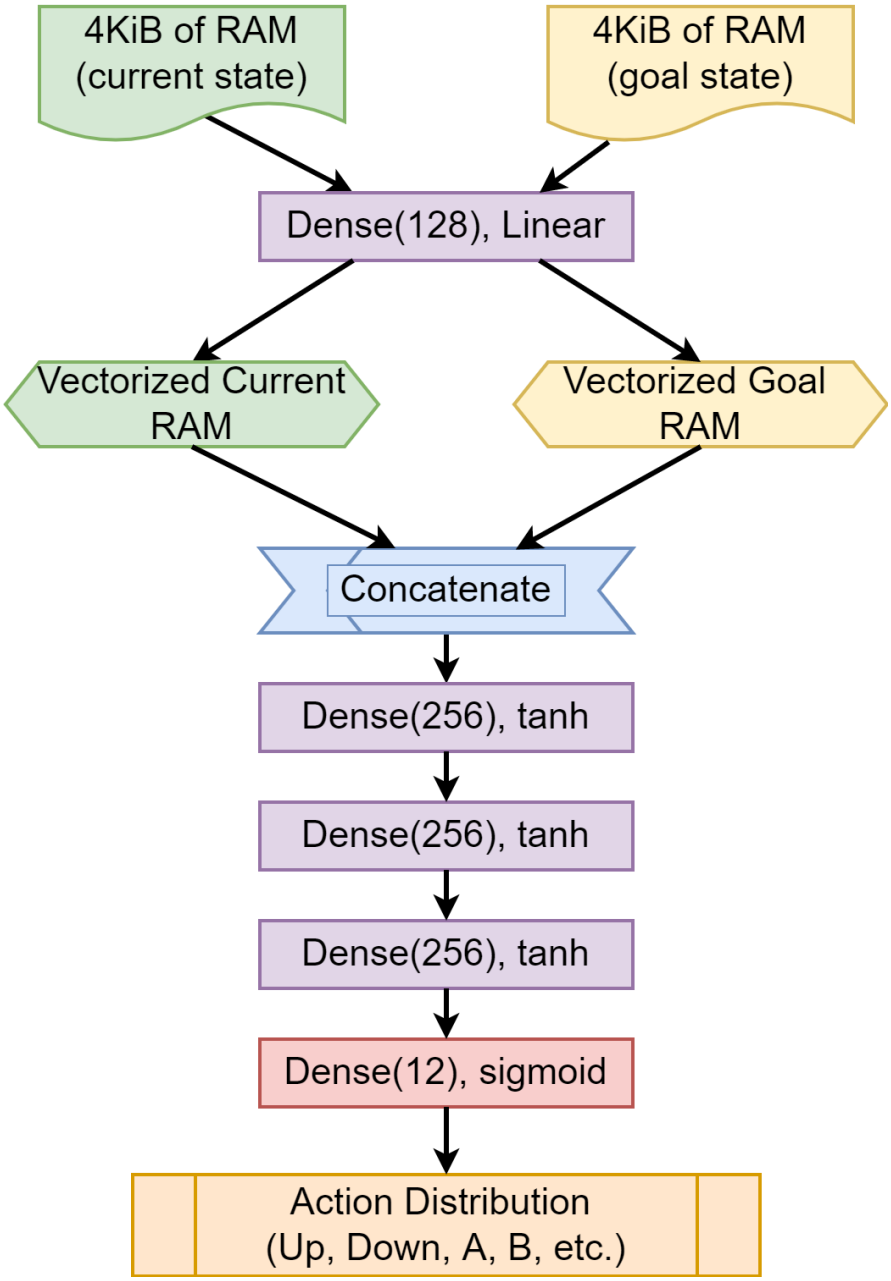


Figure 6.4: The Turbocharger NN learns how to predict a button press distribution given a current RAM vector and goal RAM vector. Doing so, we can imitate how a player moves around a playable space since we save branches of gameplay that contain starting and ending RAM states.

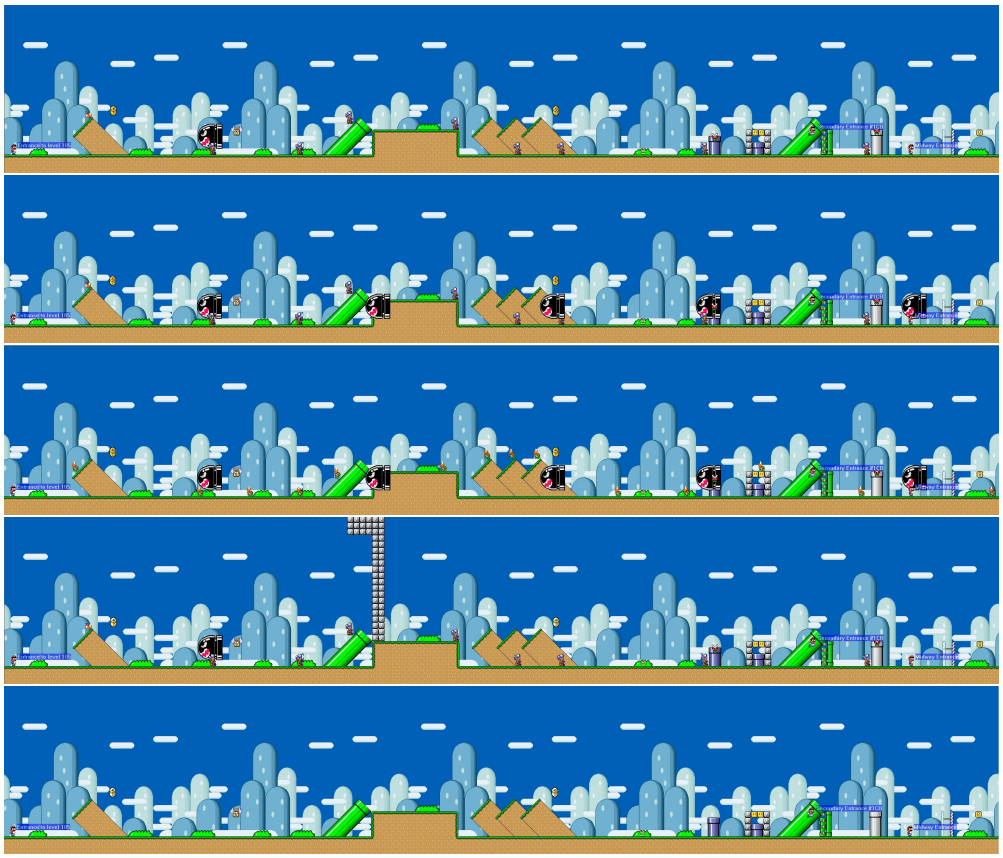


Figure 6.5: Previews of the 4 incremental versions of Super Mario World, as well as an unmodified Super Mario World level. From top to bottom, they are: Unmodified, Banzai 1, Banzai 2, Impossible, and Trivial.

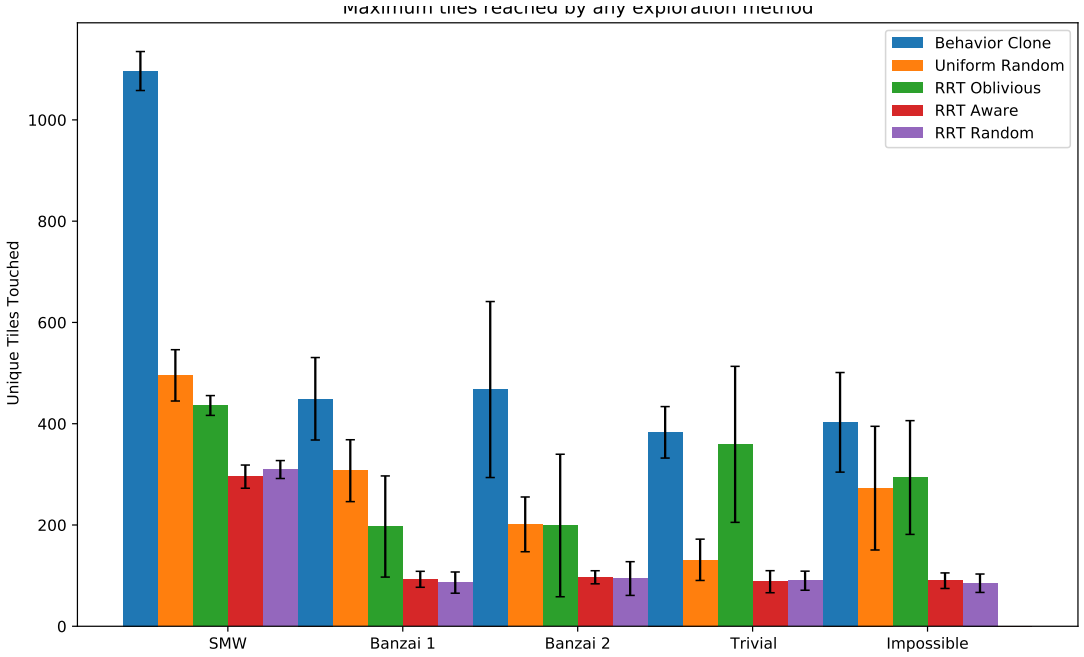


Figure 6.6: Behavior cloning clearly dominates in exploration quality, showing that RRT was actually a hindrance to exploration.

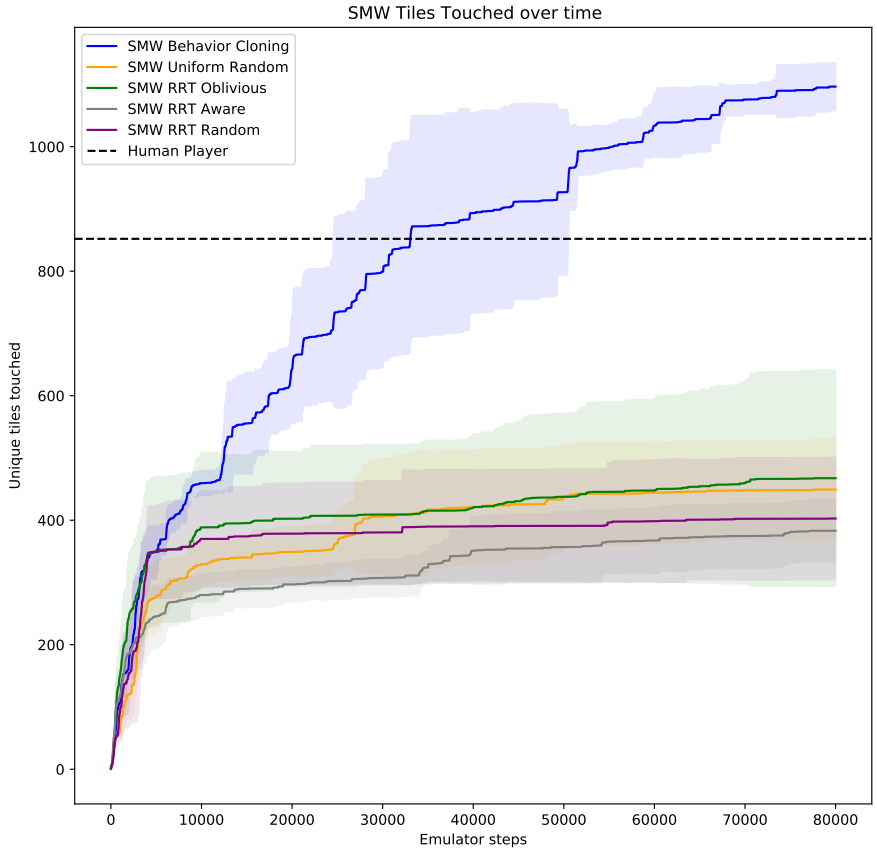
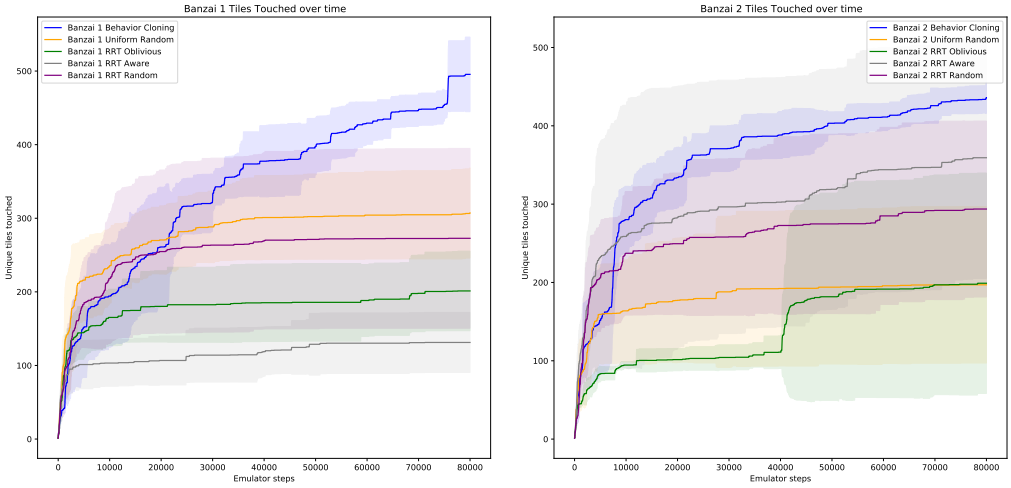
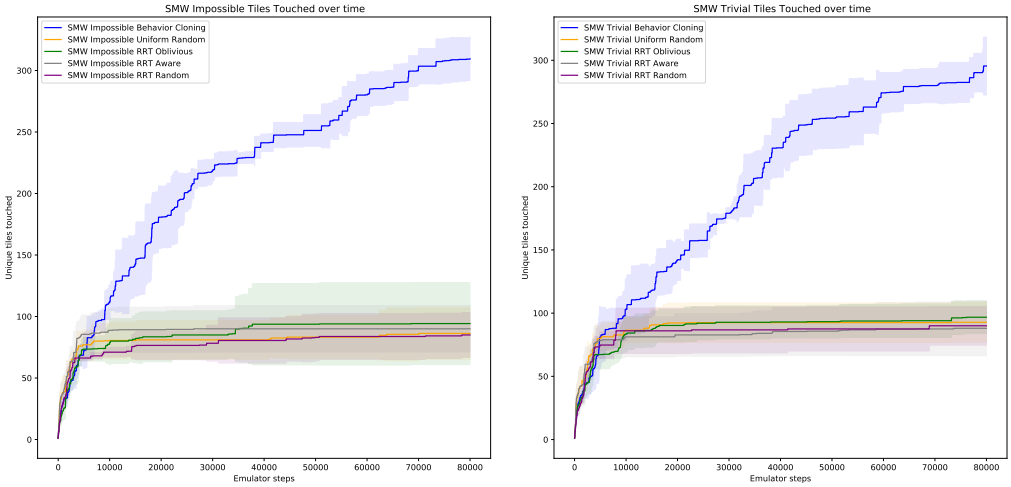


Figure 6.7: In unmodified Super Mario World, an exploration agent that acts like a human player (but not replaying button presses) explores better than what a person did in far less time.



(a) Banzai 1 (b) Banzai 2

Figure 6.8: RRT with goal-aware action selection produces inferior exploration quality compared to cloning the player behavior at specific game states. Weaknesses in game state representation and action selection are becoming apparent.



(a) Banzai 1 (b) Banzai 2

Figure 6.9: When egregious game design changes are introduced, we see a wall in exploration hit by all exploration techniques. Note that player behavior cloning still out-performs all other exploration techniques.

Chapter 7

Conclusion

While a lot has been achieved in the scope of this disseration, it is clear that a lot of research avenues have been opened. While it would be (now) incorrect to say that industry has ignored much of the academic work in automated playtesting, I would like to share some paths of research that I believe could yield fruitful progress.

Getting information out of a game to understand what is going on in it is deviously difficult and complex. Games are designed to be understandable so players can imagine themselves in game as capable of taking steps towards goals in another location. It is very easy to underestimate the challenge of representing game states as locations in intuitive space because of this intuition. Humans win games against humans because they understand the state of the game better than their opponents. And, right now, the AIs that best understand the state of the game use that knowledge to find the best ways to win, which is a single goal. There remains a body of work to translate a game's understanding into discovery algorithms, where winning is only one goal among

many.

Any tool that does automated playtesting should resemble a spellchecker, not an MRI machine. We want tired and busy developers to catch gameplay bugs quickly without getting distracted. We should be willing to accept a lot of inaccuracy in our system's reports so long as those reports get to the developer in a timely and appropriate manner. Things that are lightweight, less accurate but deliver even a little amount of actionable feedback should be pursued in lieu of perfect testing tools.

Bibliography

[1] Khalid Alemerien and Kenneth Magel. Examining the effectiveness of testing coverage tools: An empirical study. *International journal of Software Engineering and its Applications*, 8(5):139–162, 2014.

[2] Paul Ammann and Jeff Offutt. *Introduction to software testing*. Cambridge University Press, 2016.

[3] Robert C Armstrong, Ratish J Punnoose, Matthew H Wong, and Jackson R Mayo. Survey of existing tools for formal verification. *SANDIA REPORT SAND2014-20533*, 2014.

[4] Cornelius Aschermann, Sergej Schumilo, Ali Abbasi, and Thorsten Holz. Ijon: Exploring deep state spaces via fuzzing. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1597–1612. IEEE, 2020.

[5] Algirdas Avizienis, J-C Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE transactions on dependable and secure computing*, 1(1):11–33, 2004.

[6] Aaron Bauer and Zoran Popović. RRT-based game level analysis, visualization, and visual refinement. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 2012.

[7] Aaron William Bauer, Seth Cooper, and Zoran Popovic. Automated redesign of local playspace properties. In *FDG*, pages 190–197, 2013.

[8] Marc Bellemare, Sriram Srinivasan, Georg Ostrovski, Tom Schaul, David Saxton, and Remi Munos. Unifying count-based exploration and intrinsic motivation. *Advances in neural information processing systems*, 29, 2016.

[9] Antonia Bertolino. Software testing research: Achievements, challenges, dreams. In *Future of Software Engineering (FOSE'07)*, pages 85–103. IEEE, 2007.

[10] Igor Borovikov, Yunqi Zhao, Ahmad Beirami, Jesse Harder, John Kolen, James Pestrak, Jervis Pinto, Reza Pourabolghasem, Harold Chaput, Mohsen Sardari, et al. Winning isn't everything: Training agents to playtest modern games. In *AAAI Workshop on Reinforcement Learning in Games*, 2019.

[11] Cameron B Browne, Edward Powley, Daniel Whitehouse, Simon M Lucas, Peter I Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A survey of Monte Carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):1–43, 2012.

[12] Yuri Burda, Harri Edwards, Deepak Pathak, Amos Storkey, Trevor Darrell, and

Alexei A Efros. Large-scale study of curiosity-driven learning. *arXiv preprint arXiv:1808.04355*, 2018.

[13] Brian Burg, Richard Bailey, Amy J Ko, and Michael D Ernst. Interactive record/replay for web application debugging. In *Proceedings of the 26th annual ACM symposium on User interface software and technology*, pages 473–484, 2013.

[14] Mike Burghart. Evolving test in the video game industry. *Gamasutra*, June 2014.

[15] Kenneth Chang, Batu Aytemiz, and Adam M. Smith. Reveal-more: Amplifying human effort in quality assurance testing using automated exploration. In *2019 IEEE Conference on Games (CoG)*, pages 1–8, 2019.

[16] Kenneth Chang and Adam Smith. Differentia: Visualizing incremental game design changes. *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 16(1):175–181, Oct. 2020.

[17] Maxime Chevalier-Boisvert, Lucas Willems, and Suman Pal. Minimalistic gridworld environment for openai gym. `https://github.com/maximecb/gym-minigrid`, 2018.

[18] Shauvik Roy Choudhary, Alessandra Gorla, and Alessandro Orso. Automated test input generation for android: Are we there yet?(e). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 429–440. IEEE, 2015.

[19] Terry Coatta, Michael Donat, and Jafar Husain. Automated QA testing at EA: Driven by events. *Queue*, 12(5):20–10, 2014.

[20] Jeanne Collins. Conducting in-house play testing. *Gamasutra*, July 1997.

[21] Michael Cook and Azalea Raad. Hyperstate space graphs for automated game analysis. In *2019 IEEE Conference on Games (CoG)*, pages 1–8. IEEE, 2019.

[22] Casey DeWitt. On Quality Assurance. *Gamasutra*, September 2018.

[23] Yiming Ding, Carlos Florensa, Mariano Phielipp, and Pieter Abbeel. Goal-conditioned imitation learning. *arXiv preprint arXiv:1906.05838*, 2019.

[24] Joris Dormans. Simulating mechanics to study emergence in games. *Artificial Intelligence in the Game Design Process*, 2(6.2):5–2, 2011.

[25] Vijay D'silva, Daniel Kroening, and Georg Weissenbacher. A survey of automated techniques for formal software verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(7):1165–1178, 2008.

[26] Yan Duan, John Schulman, Xi Chen, Peter L. Bartlett, Ilya Sutskever, and Pieter Abbeel. Rl2: Fast reinforcement learning via slow reinforcement learning. *CoRR*, abs/1611.02779, 2016.

[27] Adrien Ecoffet, Joost Huizinga, Joel Lehman, Kenneth O. Stanley, and Jeff Clune. Go-explore: a new approach for hard-exploration problems. *CoRR*, abs/1901.10995, 2019.

[28] Adrien Ecoffet, Joost Huizinga, Joel Lehman, Kenneth O Stanley, and Jeff Clune. Go-explore: a new approach for hard-exploration problems. *arXiv preprint arXiv:1901.10995*, 2019.

[29] Wikipedia Editors. Pokemon (video game series). `https://en.wikipedia.org/wiki/Pok%C3%A9mon_(video_game_series)`, 2022.

[30] Jordan Fisher. How to make insane, procedural platformer levels, May 2012.

[31] Martin Fowler. Continuous integration. `https://martinfowler.com/articles/continuousIntegration.html`, May 2006.

[32] Jamie Fristrom. Manager in a strange land: Churning. *Gamasutra*, December 2003.

[33] Tracy Fullerton. *Game design workshop: a playcentric approach to creating innovative games*. CRC press, 2014.

[34] Pablo García-Sánchez, Alberto Tonda, Antonio M Mora, Giovanni Squillero, and Juan Julián Merelo. Automated playtesting in collectible card games using evolutionary algorithms: A case study in hearthstone. *Knowledge-Based Systems*, 153:133–146, 2018.

[35] Rachel Gillet. Inside the 'dream job' of a video game tester. *Business Insider*, June 2015.

[36] Jose Luis González Sánchez, Natalia Padilla Zea, and Francisco L. Gutiérrez. From usability to playability: Introduction to player-centred video game development

process. In Masaaki Kurosu, editor, *Human Centered Design*, pages 65–74, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[37] Cristina Guerrero-Romero, Annie Louis, and Diego Perez-Liebana. Beyond playing to win: Diversifying heuristics for GVGAI. In *2017 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 118–125. IEEE, 2017.

[38] Leopold Haller and Hernan Moraldo. https://ai.googleblog.com/2021/06/quickly-training-game-playing-agents.html, June 2021.

[39] Peter Henderson, Riashat Islam, Philip Bachman, Joelle Pineau, Doina Precup, and David Meger. Deep reinforcement learning that matters. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[40] Michael Hilton, Nicholas Nelson, Timothy Tunnell, Darko Marinov, and Danny Dig. Trade-offs in continuous integration: assurance, security, and flexibility. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 197–207, 2017.

[41] Geoffrey E Hinton and Ruslan R Salakhutdinov. Reducing the dimensionality of data with neural networks. *science*, 313(5786):504–507, 2006.

[42] Christoffer Holmgård, Michael Cerny Green, Antonios Liapis, and Julian Togelius. Automated playtesting with procedural personas through MCTS with evolved heuristics. *arXiv preprint arXiv:1802.06881*, 2018.

[43] Christoffer Holmgård, Antonios Liapis, Julian Togelius, and Georgios N Yannakakis. Evolving personas for player decision modeling. In *2014 IEEE Conference on Computational Intelligence and Games*, pages 1–8. IEEE, 2014.

[44] Ming Huo, June Verner, Liming Zhu, and Muhammad Ali Babar. Software quality and agile methods. In *Proc. of the 28th Annual International Computer Software and Applications Conference, 2004. COMPSAC 2004.*, pages 520–525. IEEE, 2004.

[45] Mikhail Jacob, Sam Devlin, and Katja Hofmann. "it's unwieldy and it takes a lot of time." challenges and opportunities for creating agents in commercial games, 2020.

[46] Alexander Jaffe, Alex Miller, Erik Andersen, Yun-En Liu, Anna Karlin, and Zoran Popović. Evaluating competitive game balance with restricted play. In *Proceedings of the Eighth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, AIIDE'12, page 26–31. AAAI Press, 2012.

[47] Anna Jenelius. The Respect QA Deserves, or Finding the Right People and Making Them Stay, April 2014.

[48] Ranjit Jhala and Rupak Majumdar. Software model checking. *ACM Computing Surveys (CSUR)*, 41(4):1–54, 2009.

[49] Oleksandra Keehl and Adam M Smith. Monster carlo: an mcts-based framework

for machine playtesting unity games. In *2018 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 1–8. IEEE, 2018.

[50] Oleksandra Keehl and Adam M Smith. Monster carlo 2: Integrating learning and tree search for machine playtesting. In *2019 IEEE Conference on Games (CoG)*, pages 1–8. IEEE, 2019.

[51] Oleksandra Keehl and Adam M Smith. Monster carlo 2: Integrating learning and tree search for machine playtesting. In *2019 IEEE Conference on Games (CoG)*, pages 1–8. IEEE, 2019.

[52] Ahmed Khalifa, Philip Bontrager, Sam Earle, and Julian Togelius. Pcgrl: Procedural content generation via reinforcement learning. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 16, pages 95–101, 2020.

[53] Ji Hun Kim and Richard Wu. Leveraging machine learning for game development, March 2021.

[54] Mathieu Lachance. How much people, time and money should QA take? part 1, January 2016.

[55] Sauli Laitinen. Better games through usability evaluation and testing. *Game Developer Informa (Gamasutra)*, June 2005.

[56] Steven M. Lavalle. Rapidly-exploring random trees: A new tool for path planning. Technical report, Iowa State University, 1998.

[57] Chris Lewis. Zenet: Generating and enforcing real-time temporal invariants. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2*, pages 329–330, 2010.

[58] Chris Lewis, Jim Whitehead, and Noah Wardrip-Fruin. What went wrong: a taxonomy of video game bugs. In *Proceedings of the fifth international conference on the foundations of digital games*, pages 108–115, 2010.

[59] Earl Logan Jr. *Handbook of turbomachinery*. CRC Press, 2003.

[60] Laurens van der Maaten and Geoffrey Hinton. Visualizing data using t-SNE. *Journal of machine learning research*, 9(Nov):2579–2605, 2008.

[61] Leland McInnes, John Healy, and James Melville. UMAP: Uniform manifold approximation and projection for dimension reduction. *arXiv preprint arXiv:1802.03426*, 2018.

[62] Stephan Merz. Model checking: A tutorial overview. *Summer School on Modeling and Verification of Parallel Processes*, pages 3–38, 2000.

[63] Mike. Mike's RPG center, 2000–2019.

[64] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis.

Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 02 2015.

[65] Emerson Murphy-Hill, Thomas Zimmermann, and Nachiappan Nagappan. Cowboys, ankle sprains, and keepers of quality: How is video game development different from software development? In *Proceedings of the 36th International Conference on Software Engineering*, pages 1–11, 2014.

[66] Alfredo Nantes, Ross Brown, and Frederic Maire. A framework for the semi-automatic testing of video games. In *Proc. of the AAAI Conference on Artificial Intelligence in Digital Entertainment (AIIDE)*, 2008.

[67] Mark J Nelson. Game metrics without players: Strategies for understanding game artifacts. In *Workshops at the Seventh Artificial Intelligence and Interactive Digital Entertainment Conference*, 2011.

[68] Monty Newborn. *Kasparov versus Deep Blue: Computer chess comes of age*. Springer Science & Business Media, 2012.

[69] Andrew Y Ng, Michael I Jordan, and Yair Weiss. On spectral clustering: Analysis and an algorithm. In *Advances in neural information processing systems*, pages 849–856, 2002.

[70] Alex Nichol, Vicki Pfau, Christopher Hesse, Oleg Klimov, and John Schulman. Gotta Learn Fast: A New Benchmark for Generalization in RL. *arXiv preprint arXiv:1804.03720*, 2018.

[71] Nightcrawler, Neil, KaioShin, RedComet, Dragonsbrethren, and Suzaku. romhacking.net, 2005–2019.

[72] Nintendo, Super Nintendo Entertainment System. Super Mario World, 1990.

[73] Nintendo, Super Nintendo Entertainment System. The Legend of Zelda: A Link To The Past, 1991.

[74] Alex Nodet. Human-like playtesting with deep learning. *Medium*, 2019.

[75] OpenAI. Openai gym, 2016.

[76] OpenAI. Gym retro, 2017.

[77] Takayuki Osa, Joni Pajarinen, Gerhard Neumann, J. Andrew Bagnell, Pieter Abbeel, and Jan Peters. An algorithmic perspective on imitation learning. *Foundations and Trends in Robotics*, 7(1–2):1–179, 2018.

[78] Joseph C. Osborn, Adam Summerville, and Michael Mateas. Automatic mapping of NES games with mappy. *CoRR*, abs/1707.03908, 2017.

[79] Nathan Partlan, Elin Carstensdottir, Erica Kleinman, Sam Snodgrass, Casper Harteveld, Gillian Smith, Camillia Matuk, Steven C Sutherland, and Magy Seif El-Nasr. Evaluation of an automatically-constructed graph-based representation for interactive narrative. In *Proceedings of the 14th International Conference on the Foundations of Digital Games*, pages 1–9, 2019.

[80] Diego Perez-Liebana, Jialin Liu, Ahmed Khalifa, Raluca D Gaina, Julian Togelius, and Simon M Lucas. General video game AI: a multi-track framework for evaluating agents, games and content generation algorithms. *arXiv preprint arXiv:1802.10363*, 2018.

[81] James L Peterson. Petri nets. *ACM Computing Surveys (CSUR)*, 9(3):223–252, 1977.

[82] Johannes Pfau, Jan David Smeddinck, and Rainer Malaka. Automated Game Testing with ICARUS: Intelligent Completion of Adventure Riddles via Unsupervised Solving. In *Extended Abstracts Publication of the Annual Symposium on Computer-Human Interaction in Play (CHI Play)*, pages 153–164, 2017.

[83] Bruce Potter and Gary McGraw. Software security testing. *IEEE Security & Privacy*, 2(5):81–85, 2004.

[84] Roberta Raileanu and Tim Rocktäschel. Ride: Rewarding impact-driven exploration for procedurally-generated environments. In *International Conference on Learning Representations*, 2019.

[85] Stéphane Ross, Geoffrey Gordon, and Drew Bagnell. A reduction of imitation learning and structured prediction to no-regret online learning. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pages 627–635. JMLR Workshop and Conference Proceedings, 2011.

[86] Tom Schaul. A video game description language for model-based or interac-

tive learning. In *2013 IEEE Conference on Computational Inteligence in Games (CIG)*, pages 1–8. IEEE, 2013.

[87] Jesse Schell. *The Art of Game Design: A book of lenses*. CRC press, 2008.

[88] Jason Schreier. Quality Assured: What It's Really Like To Test Games For A Living, January 2017.

[89] Ben Shneiderman. The eyes have it: A task by data type taxonomy for information visualizations. In *Proceedings 1996 IEEE symposium on visual languages*, pages 336–343. IEEE, 1996.

[90] Charles M. Shultz. *Game Testing All In One*. Thomson Course Technology PTR, 2005.

[91] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489, 2016.

[92] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of go without human knowledge. *nature*, 550(7676):354–359, 2017.

[93] Adam Smith and Michael Mateas. Computational caricatures: Probing the game

design process with AI. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 2011.

[94] Adam M Smith. Open problem: Reusable gameplay trace samplers. In *Ninth Artificial Intelligence and Interactive Digital Entertainment Conference*, 2013.

[95] Adam M Smith, Mark J Nelson, and Michael Mateas. Computational Support for Play Testing Game Sketches. In *Proc. of the AAAI Conference on Artificial Intelligence in Interactive Entertainment (AIIDE)*, 2009.

[96] Adam M Smith, Mark J Nelson, and Michael Mateas. Ludocore: A logical game engine for modeling videogames. In *Proceedings of the 2010 IEEE Conference on Computational Intelligence and Games*, pages 91–98. IEEE, 2010.

[97] Kenneth O. Stanley. `https://www.infoq.com/presentations/Searching-Without-Objectives/`, 2010.

[98] M-A Storey, Kenny Wong, F David Fracchia, and Hausi A Muller. On integrating visualization techniques for effective software exploration. In *Proceedings of VIZ'97: Visualization Conference, Information Visualization Symposium and Parallel Rendering Symposium*, pages 38–45. IEEE, 1997.

[99] Ari Takanen, Jared D Demott, Charles Miller, and Atte Kettunen. *Fuzzing for software security testing and quality assurance*. Artech House, 2018.

[100] Rebekah Valentine. Turns Out The Hardest Part of Making a Game Is...Everything, August 2021.

[101] Jan van Valburg. Automated testing and profiling for call of duty, 2018.

[102] Oriol Vinyals, Igor Babuschkin, Junyoung Chung, Michael Mathieu, Max Jaderberg, Wojciech M. Czarnecki, Andrew Dudzik, Aja Huang, Petko Georgiev, Richard Powell, Timo Ewalds, Dan Horgan, Manuel Kroiss, Ivo Danihelka, John Agapiou, Junhyuk Oh, Valentin Dalibard, David Choi, Laurent Sifre, Yury Sulsky, Sasha Vezhnevets, James Molloy, Trevor Cai, David Budden, Tom Paine, Caglar Gulcehre, Ziyu Wang, Tobias Pfaff, Toby Pohlen, Yuhuai Wu, Dani Yogatama, Julia Cohen, Katrina McKinney, Oliver Smith, Tom Schaul, Timothy Lillicrap, Chris Apps, Koray Kavukcuoglu, Demis Hassabis, and David Silver. AlphaStar: Mastering the Real-Time Strategy Game StarCraft II, 2019.

[103] David Gray Widder, Michael Hilton, Christian Kästner, and Bogdan Vasilescu. A conceptual replication of continuous integration pain points in the context of travis ci. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, page 647–658, New York, NY, USA, 2019. Association for Computing Machinery.

[104] David Wilson. Quality quality assurance: A methodology for wide-spectrum game testing. *Gamasutra*, April 2009.

[105] Filip Wiltgren. A simple way to get great playtesting feedback. *Gamasutra*, February 2016.

[106] Gang Xiao, Finnegan Southey, Robert C Holte, and Dana Wilkinson. Software testing by active learning for commercial games. In *AAAI 2005*, pages 898–903, 2005.

[107] Michał Zalewski. american fuzzy lop. `https://lcamtuf.coredump.cx/afl/`, 2022.

[108] Zeping Zhan, Batu Aytemiz, and Adam M Smith. Taking the scenic route: Automatic exploration for videogames. *Proc. of the 2nd Workshop on Knowledge Extraction from Games co-located with 33rd AAAI Conference on Artificial Intelligence (AAAI 2019)*, 2018.

[109] Zeping Zhan and Adam M Smith. Retrieving game states with moment vectors. In *AAAI Workshops*, pages 586–592, 2018.

[110] Yangyang Zhao, Alexander Serebrenik, Yuming Zhou, Vladimir Filkov, and Bogdan Vasilescu. The impact of continuous integration on other software development practices: a large-scale empirical study. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 60–71. IEEE, 2017.

[111] Alexander Zook, Eric Fruchter, and Mark O Riedl. Automatic playtesting for game parameter tuning via active learning. *arXiv preprint arXiv:1908.01417*, 2019.

Appendix A

Code Artifacts

All source code and project data (including modified game roms and gameplay traces) for Reveal-More, Differentia, and Turbocharger can be found in the following repositories:

- Reveal-More:

  `https://bitbucket.org/kenneth_chang/reveal-more-redux/src/master/`

- Differentia:

  `https://bitbucket.org/kenneth_chang/whatifyoucoulddiff/src/master/`

- Turbocharger:

  `https://bitbucket.org/kenneth_chang/turbocharger/src/master/`