THE EVOLUTION OF FUN:
A GENERIC MODEL OF VIDEO GAME CHALLENGE FOR AUTOMATIC LEVEL DESIGN

by

Nathan Sorenson

B.Sc.Hons., Computer Science, University of Calgary, 2008

B.Sc., General Mathematics, University of Calgary, 2008

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
in the School
of
Interactive Arts and Techology

SIMON FRASER UNIVERSITY
Fall 2010

APPROVAL

Name: Nathan Sorenson

Degree: Master of Science

Title of thesis: The Evolution of Fun: A Generic Model of Video Game Challenge for Automatic Level Design

Examining Committee:

Chair:

Dr. John Bowes
Professor,
School of Interactive Arts and Technology

Steve DiPaola
Senior Supervisor
Associate Professor,
School of Interactive Arts and Technology

Dr. Philippe Pasquier
Supervisor
Assistant Professor,
School of Interactive Arts and Technology

Dr. Marek Hatala
External Examiner
Associate Professor,
School of Interactive Arts and Technology

Date Approved: 2010-12-08

Declaration of
Partial Copyright Licence

Abstract

This thesis presents an approach to automatic video game level design consisting of a novel computational model of player enjoyment and a generative system based on evolutionary computing. The model is grounded in player experience research and game design theory and is used to estimate the entertainment value of game levels as a function of their constituent rhythm groups: alternating periods of high and low challenge. In comparison to existing, bottom-up techniques such as rule-based systems, the model affords a number of distinct advantages: it can be generalized to different types of games; it provides adjustable parameters representing semantically meaningful concepts such as difficulty and player skill; and it can facilitate mixed-initiative collaboration between the automated system and a human designer. The generative system represents a unique combination of genetic algorithms and constraint solving methods and leverages the model to create fun levels for two different games.

Acknowledgments

I would like to thank my senior supervisor, Steve DiPaola, for his enthusiasm and guidance, and for cultivating my interest in evolutionary computation and generative design. I must also express gratitude toward my supervisor, Philippe Pasquier, whose continuous involvement, feedback, and support have had a tremendous impact on the development of this research.

Thanks to all of the members of the iViz lab (ivizlab.sfu.ca) and the MAMAS lab (metacreation.net) who provided valuable feedback on various papers and presentations related to my work. As well, I would like to thank Rich Hickey and the Clojure (clojure.org) community for creating a productive and beautiful programming language, and also the authors of the JaCoP (jacop.osolpro.com) constraint satisfaction library, which has proven indispensible. I also greatly appreciate the efforts of Noor Shaker, Julian Togelius, and Georgios Yannakakis in organizing and overseeing the 2010 Mario AI Championship. I am also grateful for the work of Markus Persson in creating *Infinite Mario*.

Thanks to my family, for their constant love and encouragement; to Jay McCarrol and Matt Johnson for motivational support during times of writer's block; and most of all, to my wife, Megan, for her unfailing dedication, patience, and optimism. Her various contributions as companion, editor, and videogame opponent have been incalculable, and without her this thesis could not have been completed.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The videogame industry continues to grow and increasing competition is driving game developers to seek novel ways of expending less effort to create more content. In addition, larger and more diverse gaming audiences increase the demand for more personalized and varied play experiences. As a result of these growing concerns, there has been interest in the use of computational generative techniques known as procedural content creation. These techniques limit the effort required of a human designer by producing some game content automatically through various computational approaches, such as rule-based systems, grammars, fractals, and genetic algorithms. If a generative technique can be implemented with less effort than is required for creating content by hand, then the potential amount of content is greatly increased. Generated content not only requires less effort to produce, but also has the potential of greater adaptability. If the production of the content can be controlled by external parameters in a predictable way, the content of a game can be adapted to a wide variety of preferences, for example, in the creation of easier environments for novice players.

This thesis describes a generic approach to procedural content generation. Our system automatically constructs game environments (levels) according to an explicit model of challenge-based fun, which, put simply, is the enjoyment that results from completing game challenges that are neither too difficult, nor too easy. This approach is an example of a top-down architecture that stands in contrast to existing techniques, which are typically ad-hoc, rule-based systems. Such rule-based systems lack an explicit model of the desirable characteristics of level design, are difficult to construct, and cannot be easily generalized to

multiple game contexts. As well, such systems, if they offer explicit customization parameters at all, generally provide content authors with unpredictable, high-level control over the output, with a non-obvious relationship between the system parameters and the generated levels [39].

The model we develop draws an explicit connection between variations of challenge over time and the amount of enjoyment experienced by players. We focus on recognizable structures known as "rhythm groups" [46], which are alternating patterns of high and low difficulty. These structures are evident in professionally-authored videogame levels, and represent a particularly effective way to engage players. As a result of our focus on automated level design, we consider games where the challenge experienced by the player is primarily a function of the level design, since our goal is to create environments that exhibit engaging challenge dynamics. This means, as well, that we do not consider broader dimensions of fun in videogames, such as the topics relating to narrative or exploration; we focus on fun only as it pertains to the experience of overcoming challenge.

Our generative system, which is responsible for the automatic creation of new levels, assumes a top-down architecture through the use of evolutionary computation. With this approach, possible level designs are treated as individuals in a population, and individuals that are deemed to be highly fit—which, in our case, means that they display appropriate difficulty dynamics—are able to pass their unique characteristics to future generations of level designs through their genetic code. This can be seen as a top-down approach because levels are evaluated in terms of a high-level fitness function without regard to the particular technique that generated them. This feature affords our system the distinct advantage of being generalizable and customizable at a high level.

The underlying model of fun guides the output of our generative system and functions as a high-level evaluator of level design quality. In other words, it serves as our evolutionary system's fitness function: the model analyzes the particular arrangement of level elements and returns a single value estimating that level's quality. The model is intended to be clear and parsimonious, expressing important level design principles in a manner that can be easily inspected. This fact enables the model to provide designers with parameters that have a direct and intuitive relationship to the system output. For example, we provide a single model parameter that corresponds to player skill; by manipulating this parameter, the system can generate levels with higher or lower overall degrees of difficulty. Finally, it is a generic model, in that it can be applied to different game contexts and is not defined

in terms of any particular game mechanic or any specific generative techniques.

1.1 Motivation

Commercial games are exhibiting ever-increasing amounts of content in the form of animated characters, dialog, game items, and environments. For example, the ongoing profitability of massively multi-player on-line games such as *World of Warcraft* [36] depends on providing players with an increasing supply of locations, quests, and collectible items, all of which must be authored by human designers. Many AAA titles command budgets of tens of millions of dollars, or even, as in the case of *Grand Theft Auto 4*, 100 million dollars [8]. Such monumental efforts are well beyond the reach of smaller game developers that lack the resources to produce content at this scale. Automatic level design promises to reduce the high cost of creating game levels.

As well, with the success of non-traditional forms of interaction such as the *Wii* and the exponential growth of new player markets through the popularity of casual games as typified by companies such as Zynga and Playfish, there has never before been as diverse an audience for videogames. The ability to adapt game content to the wide range of skill sets would be invaluable in reaching these broad and eclectic groups.

1.2 Thesis Overview

The central contributions of this thesis are as follows:

- *A computational model of challenge-based fun.* The model estimates the entertainment value of levels for "challenge-based" videogames. We clearly define the scope of the model, and describe our usage of the term 'challenge-based fun' as the response to game structures that present challenges that are neither too difficult nor too easy; levels in challenge-based games are judged to be the most entertaining when they are not too difficult to complete, but also not so easy as to lose the player's interest. We validate this characterization of level quality against real-world levels, and apply it to two different challenge-based games to demonstrate its generality.

- *An automatic level generation system.* Our generative system creates levels that exhibit good challenge dynamics, as specified by our computational model. Our approach

consists of a novel hybrid between genetic algorithms and constraint solvers, and is able to automatically create new videogame levels without requiring direct human involvement.

We emphasize that our approach is novel among procedural game content generation system in that it simultaneously satisfies the following properties:

1. *It is generalizable.* We can apply both our model and our generative system to evaluate level quality and design new levels for various game genres.

2. *It provides semantically meaningful parameters.* Our model offers a clear way to account for players of various skill levels, through the manipulation of parameters that have a predictable effect on the levels created by the system. This is necessary because challenge dynamics that are enjoyable for highly skilled players would certainly not be enjoyable for novices.

3. *It facilitates mixed-initiative design.* Although our system does not require the input of a human designer, it does allow for designers to create portions of game levels manually. It then creates procedural content that seamlessly integrates with the human-designed content. This feature is particularly difficult for naïve, rule-based approaches to replicate, as such techniques are not able to reflect on how the procedurally generated content will interact with the human-designed content.

Chapter 2 covers background material regarding theoretical accounts of fun in videogames, as well as evolutionary algorithms, constraint satisfaction methods, and current work regarding procedural content generation for videogames. Chapter 3 introduces our model of player enjoyment, which drives the rest of the generative system. We describe the scope of the model, outlining how it applies specifically to challenge-based games. We then justify the model's design, drawing from videogame literature as well as an original analysis of existing commercial game levels. We evaluate this model by its effectiveness of identifying commercial-quality levels among arbitrarily generated levels. Chapter 4 discusses the design of the generative system, which produces the game levels. We show how our model is used as a fitness function in an evolutionary system, where a population of possible level designs is evolved to maximize level quality. We describe how we represent levels as a collection of *design elements* (DEs), and how we ensure that level designs satisfy playability constraints.

We then apply the system to two different game contexts, demonstrating its general applicability. Chapter 5 documents the application of the system to the game *Super Mario Bros.* [32], and Chapter 6 demonstrates an application of the system to a simplified version of *The Legend of Zelda* [31]. Chapter 7 concludes with a discussion of the benefits of this approach and outlines future work.

Chapter 2

Background

Our work builds upon research from two distinct areas. The generative system itself, which produces the actual level designs, relies on evolutionary computation techniques and constraint satisfaction methods. Our model of fun, which guides the output of the generative system by estimating the entertainment value of potential level designs, is constructed according to both theorizations of fun in videogames and analyses of actual levels found in commercial titles.

2.1 Characterizations of Fun

Fun is a complex psychological phenomenon that does not admit a single, overarching definition. For this reason, it is necessary to clearly delineate the scope of any attempts to define this concept. We are particularly concerned with fun as a result of videogame playing. However, this is domain is still very broad; Hunicke, LeBlanc, and Zubek [21] even argue that the term 'fun' should be discarded entirely, as it is too vague to be considered a practical unit of analysis. They suggest eight more precise terms: "challenge," "discovery," "fantasy," "narrative," "sensation," "expression," "fellowship" and "submission." Similar categories are offered by other authors in their discussions of the nature of fun in play. Malone [27] identifies three principal components to fun in videogames, "challenge," "curiosity," and "fantasy," while Apter [3] offers a similar list, which includes "challenge," "exploration," "fiction and narrative," "arousing stimulation," "cognitive synergy," "facing danger" and "negativism." Garneau's [16] list of fourteen forms of fun includes similar terms, but adds such categories as "physical activity," "comedy," and "power."

Although these taxonomies provide useful terminology, it is doubtful whether they allow for deeper analysis of the nature of fun in videogames. The limited structuring of their categories and the lack of any overarching model makes it difficult to extract general design principles from them. These categorizations typically identify *what* types of fun can be observed, but they do not generally specify *why* such things are fun and therefore cannot offer advice as to *how* designs can evoke a particular type of fun.

2.1.1 Flow

A common thread among analyses of game performance and player enjoyment is the notion of "flow," as expounded in the work of Csikszentmihalyi [11]. The psychological state of flow is brought about when a number of prerequisite conditions are satisfied, such as one feeling in control of a situation, losing awareness of the passage of time, and executing a task that is neither too easy nor too difficult for one's skill level. Flow is characterized by intense focus and heightened task performance and is often referred to as a state of "optimal experience." Sweetser and Wyeth have adapted the principles of this concept into a framework called "GameFlow" [51] that identifies properties of game designs which especially facilitate the creation of a sense of flow.

Several authors note that the concept of flow is valuable in understanding certain gaming experiences, particularly the connection between challenge and fun. In *A Theory of Fun*, Koster states that "fun is the act of mastering a problem mentally" [25] and that pleasure in games is ultimately generated by the process of overcoming difficult tasks, such as identifying patterns in the behaviour of enemy characters or developing the muscle-memory necessary to execute a sequence of button presses in a fighting game. If the task is too difficult, the player does not experience a sense of mastery. Conversely, if the task is too easy, the player does not need to develop any skills to succeed. Salen and Zimmerman confirm the central role of difficulty in providing fun experiences, and although they emphasize that flow is not synonymous with fun, they do claim that challenge and frustration are "essential to game pleasure" [41].

There are definite affinities between the challenge-based dimension of flow and the Yerkes-Dodson law [61], which states that performance reaches its maximum when the arousal felt during the completion of a task is neither too little nor too great. Piselli et al. [38] have shown that this phenomenon is equally applicable to the context of video

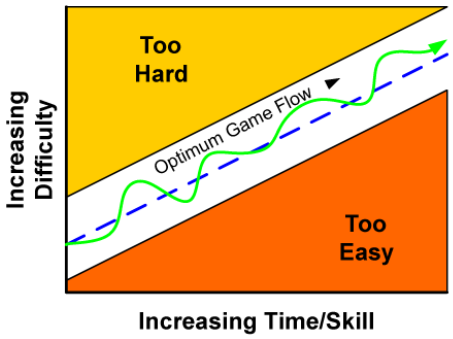


Figure 2.1: Variation in the flow channel (© 2004, Noah Falstein).

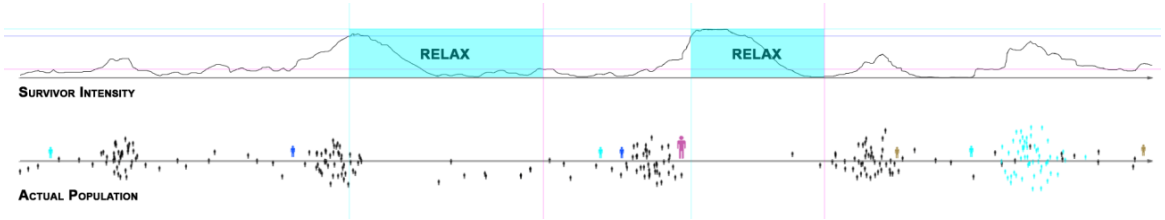


Figure 2.2: Dramatic game pacing in *Left 4 Dead* (© 2009, Valve Software).

games, even when considering pleasure, instead of task performance, as a function of difficulty; players have the most fun when presented with challenges that are difficult, but not impossible, to overcome.

Jesper Juul points out that challenge is important but should not be constant: "the desire for game balance, losing some, winning some, is also a desire for variation in the challenge and difficulty of the game" [23]. This point is illustrated clearly by Falstein [15] in Figure 2.1. According to Michael Booth, this idea can also be found in the use of Valve Software's game *Left 4 Dead*, where the "AI Director" system attempts to achieve "dramatic game pacing" by placing enemies so that there are "periods of quiet tension punctuated by moments of intense combat" [4], as illustrated in Figure 2.2. We employ the term "rhythm group," as introduced by Smith et al. [46], to refer to this recognizable structure of oscillating periods of high and low difficulty.

2.2 Evolutionary Systems

Evolutionary computation is a blanket term that refers to a number of related search techniques in the field of artificial intelligence. These techniques are inspired by the process of

natural selection and model the task of finding an optimal solution to a particular problem as a process of accumulating incrementally beneficial mutations. All evolutionary search techniques require the specification of some form of objective function to evaluate individuals, which is called the fitness function. Individuals are subjected to a continuous stochastic mutation, and alterations are deemed beneficial if they result in an individual that increases the value of the fitness function.

2.2.1 Genetic Algorithms

Our generative system is most closely related to the evolutionary computational technique known as genetic algorithms (GAs) [20]. A genetic algorithm requires the representation of a particular solution as a linear sequence of characters, known as a genotype. In its most basic specification, the alphabet of characters is the set of binary digits, 0 and 1. Every problem must then be defined in terms of a string of binary digits. To evaluate a particular individual, the genotype must be converted into an actual solution (referred to as the phenotype) in a process known as the genotype to phenotype mapping.

In addition to the genetic encoding, a genetic algorithm requires the specification of two genetic operators: a mutation and a crossover operator. Mutation involves altering a specific gene in a genotype by, for example, flipping a bit in the genotype from 0 to 1 or vice versa. Crossover, analogous to the physical process of genetic crossover in sexual reproduction, is an operator that constructs a new individual from two parents by creating a new genotype that combines genetic material from each parent. There are several different approaches to determining what genetic material from each parent contributes to the child, the simplest being single-point crossover. This operation identifies a random location in the genotype as a cut point. All the genes situated to one side of the cut point are taken from the first parent and all the genes to the other side of the cut point are taken from the other parent, and these two fragments are spliced together to form a new genotype.

Finally, a selection policy is adopted, which determines which individuals are selected for subsequent generations. A popular choice is tournament selection. A tournament size n is specified at the beginning of the run. The next generation is determined through a number of tournament rounds. Each round consists of selecting groups of n individuals at random from the current generation and of choosing the individual with the highest fitness within this group to be included in the next generation.

Because individuals are only compared against groups of n individuals as opposed to

against the entire population, there is always a chance that individuals with relatively lower fitness will be selected for the next generation. This property of occasionally selecting less-fit individuals typically enables greater optimizations; temporarily exploring variations with lower fitness can produce individuals with higher fitness than the current leaders. The fitness function can be thought of as a high-dimensional function which has its domain the space of all possible individuals. Each generation serves to sample the value of this function at a small (relative to the total number of possible individuals) number of points. By sometimes considering points with lower values, the system is able to escape the many local maxima that are present in a typical optimization problem.

Characteristics of Genetic Algorithms

Genetic algorithms are highly effective in practice. This success is partially due to their ability to escape local maxima. Although the selection of individuals of lower fitness partially contributes to this feature, the existence of the crossover operator is what makes genetic algorithms distinct from other local search techniques. This operator allows for large movements in the search space while maintaining beneficial configurations of genes in the genotype. It is this feature which makes the choice of encoding scheme critical; that is, an effective genetic coding is precisely one which behaves well under the genetic operators of mutation and crossover. Mutation should effect small movements in the search space, and crossover should preserve gene combinations that contribute significantly to an individual's fitness. This property is referred to as "gene linkage" [17] and is attained by having genes likely to interact in beneficial ways exist in close proximity to one another within the genotype so that they are less likely to be separated by the crossover's cut point. The representation we adopt for our procedural level generator deliberately attempts to maintain the important property of gene linkage.

Though exact methods such as linear programming are often more efficient than GAs, they are not applicable to many real-world tasks. GAs have proven to be effective in solving high dimensional problems that are difficult to formulate in terms compatible with an exact solution. Indeed, it is often possible to construct an objective measure of quality for concrete solutions in the form of a fitness function. Because the fitness function operates solely on phenotypes, it can be specified independently of the particular method used to map the problem domain onto its genotype representation; the task of adapting the problem space into a search space (defining a genotype encoding) is then separated from the task

of adapting the problem solutions to an objective evaluation criterion (defining a fitness function). Furthermore, approximate methods such as GAs are sometimes necessary even if an exact approach is possible, as the dimensionality of the search space is too large for the exact method to traverse.

In some applications, finding the global optimum is of less interest than the exploration of the search space itself. It is for this reason that evolutionary computation as been commonly applied to the context of computational art, as popularized by the work of Karl Sims [45]. In what is known as interactive evolution, human participants act as the fitness function by evaluating evolutionary populations by hand. Procedural game design is similar to computer generated art in that both applications are governed by largely subjective and aesthetic principles. In such domains, the notion of a globally optimal solution is not as valuable as a method for traversing an incredibly diverse and artistically rich search space [12].

2.2.2 Genetic Algorithm Extensions

Genetic algorithms are also effective because they can be significantly customized to a specific problem domain. In fact, it is critical to choose carefully the genetic representation, selection technique, and crossover and mutation rates in order to conduct a successful evolutionary run for a given problem. Sometimes, however, these customizations are not enough to ensure that the population is able to effectively explore the search space. The most predominant obstacle is premature convergence, whereby the entire population loses diversity and focuses too narrowly on exploiting a local maximum.

Several extensions to the basic GA exist to mitigate this occurrence. The island model, comprehensively documented by Tomassini [56], involves running several independent populations in parallel. Premature convergence only occurs within the confines of a single island, and the diversity of the total population is not lost. Occasional migrations are performed after a certain number of generations, allowing for cross-fertilization between the islands, and are conducted by copying a number of the highest-fitness individuals to neighbouring islands. Exactly how often these migrations occur, how many individuals are included, and how the island's topology is configured must be determined so as best to reduce premature convergence without completely isolating the islands from each other.

Another hindrance to a GA's effectiveness is the problem of discontinuity in the fitness landscape. In such a context, small alterations to the genotype can cause large changes

in the fitness of the individual, disrupting the evolutionary search. This can happen when a problem domain is highly constrained, as is often the case with level design; if, in an adventure game, a key required to open a certain door is moved just a few feet to the "wrong" side of that door, the entire level could be rendered unplayable. To address this, one can either produce a genetic encoding that is not able to express such infeasible designs, or one can actively repair or otherwise eliminate such designs from consideration through the use of complementary techniques, such as constraint satisfaction methods. To ensure the greatest generality by not pre-emptively restricting the search space, we adopt the latter approach in the present work.

2.3 Constraint Satisfaction Methods

Constraint satisfaction (CS) methods are well suited to problems that can be specified in terms of constraint relationships between finite-domain variables. Such techniques employ constraint propagation and backtracking to eliminate searching through solutions that could not possibly satisfy the stated constraints. Because many highly constrained problems are challenging to solve with an evolutionary approach, constraint satisfaction methods are often used as a complementary technique in hybrid GA/CS systems [9]. We implement such a hybrid solution for our own generative system.

We use the open source java library JaCoP [26] as a base for the implementation of our constraint satisfaction approach. It is a mature library that implements a large number of published constraints and search techniques. As well, it is designed with extensibility in mind, so additional constraints and search approaches can be added to the system. We make particular use of its geometrically-based constraint functionality, which is based on work by Carlsson et al. [7] and provides good performance on packing shapes within k-dimensional spaces. Because many level design constraints can be expressed in such terms, this functionality is beneficial to our context.

2.3.1 FI-2pop Constraint Satisfaction

Not all level design requirements can easily be enforced through such constraints, however. For example, the second game context to which we apply our generative system requires that there be a connected path between a series of rooms from a start point to an end point. In this situation, it is easier to detect a state of proper connectivity by traversing the rooms in

sequence than to express connectivity as a collection of local geometric constraints between pairs of doors and rooms. This traversal of rooms requires a completed level design, and, as such, is suited to a local search technique rather than a constraint propagation approach. If the question of connectivity were framed in terms of local search, the degree to which one could travel between the rooms would serve as the fitness function. In this case, we do not evaluate the entertainment quality of a level but instead determine what proportion of the level can be completed.

This intuition motivates the Feasible/Infeasible 2-Population genetic algorithm (FI-2pop), which we employ as a secondary stage for those problems which cannot be quickly solved with the typical constraint satisfaction methods employed by the JaCoP library. Introduced by Kimbrough et al. [24], FI-2pop is an approach of using GAs to solve the problem of having to simultaneously optimize an objective function while satisfying design constraints. This is done by maintaining two parallel populations; one is referred to as the feasible population, and the other is designated the infeasible population. The former population is evolved by a genetic algorithm that has a normal objective fitness function; however, individuals that are found to violate certain design constraints (such as a level with no path from the beginning to the end) are removed from the feasible population and placed in the infeasible population. This population is evolved according to a different fitness function. It is evaluated not on an objective function, such as 'entertainment quality,' but on how severely it violates constraints. By minimizing this measurement with the same genetic operators of crossover and mutation, the system produces individuals that do not violate any constraints. At this point, these individuals can be returned to the feasible population. Therefore, within this optimization model, one does not need to describe constraints in terms of local geometric properties; instead, one must provide a method for measuring the degree to which an individual violates constraints.

2.4 Videogame Generative Systems

2.4.1 Industrial Systems

Procedural content has been present in commercial videogames for many years. The original example of generative content comes from the dungeon crawler *Rogue* [57], which is depicted in Figure 2.3. Its characteristic randomly generated levels have influenced a large number of "roguelikes" [53, 62], including the highly-successful *Diablo* games [42]. Randomly generated

(a) *Rogue*, 1983. (b) *Freeciv*, 2010.

Figure 2.3: Procedural content creation in commercial videogames.

levels are also often seen in strategy games such as *Civilization* [28].

Because most commercial games are proprietary, it is not possible to identify the exact techniques that were used to generate their content. However, open source games, such as *Freeciv* [52] (a game similar to *Civilization* and seen in Figure 2.3) or *Oblige* [2] (an open source map generator for *Doom*-like first-person shooter games), reveal a reliance on rule-based systems comprising large switch statements and ad-hoc random values.

2.4.2 Academic Systems

Procedural content creation is an active area of research, and many different approaches exist. A particularly ambitious project has been the automatic generation of entire games. Cameron Browne [5] employs a genetic programming approach to construct combinatorial, abstract games that are similar to chess and go, with rules expressed through a restricted logic language. He successfully produces entertaining games using a fitness function that consists of a weighted sum of 57 design criteria, drawn from a wide array of sources, including psychology, subjective aesthetics, and personal communication with game designers. We also use an evolutionary approach but strive to make the underlying model more parsimonious and transparent. Indeed, a fitness function with 57 aspects may serve to produce good results in practice, but it would be difficult to translate such a function to another context, because some of the fitness features might not apply.

Togelius and Schmidhuber also explore the evolution of game designs [54]. They consider designs to be fun insofar as a neural net is able to learn to play that game, arguing that it is the process of learning to master a task which ultimately provides pleasure, based

on Schmidhuber's notion of artificial curiosity [43]. Although this method could be quite general, as it is defined in terms of artificial creativity as opposed to concrete, game-specific requirements, it does not offer a transparent model that can clearly communicate what game structures contribute toward fun. As well, the effectiveness of this approach could be limited by the degree to which neural nets are able to mirror the experience of human players. Neural nets are also used by Shaker et al. [44] to statistically predict player satisfaction in platformer games, for the purpose of adjusting parameters to a rule-based generative system.

Hastings et al. [18] also employ neural nets in evolving weapons for a space-themed videogame. However, these neural nets are not used to evaluate fitness, as in the previous two cases, but are, instead, employed to represent weapon behaviour. The behaviours are evolved according to the neuroevolution of augmenting topologies method. This approach results in remarkably diverse and complex weapon properties. The fitness of a given weapon design is inferred from the behaviour of the player as the game progresses; if the player uses a weapon frequently, similar weapons are favoured for reproduction in future generations. Conversely, if a weapon is left unused, it is considered less fit. This approach is an example of an interactive fitness function, whereby evolution is guided through human choice. Our work differs because we provide an automatic fitness function that does not require direct human involvement.

Smith et al. [48] introduce a mixed-initiative design tool for the creation of 2D platformer games. Their tool allows certain portions to be specified by hand while the rest are constructed according to a rule-based system. This system produces levels that conform to a rhythm-group structure, with contiguous periods of challenge interspersed with moments of rest. Although our work shares the goal of permitting mixed-initiative collaboration between a human and the system, we express rhythm groups explicitly through an objective model instead of implicitly through the behaviour of a set of production rules.

Chapter 3

Model of Fun

A computational model of fun in videogames is a central contribution of this thesis. This model is described in terms of rhythm groups, as introduced by Smith et al. [46], which are level design patterns that represent repeating oscillations between periods of high and low challenge. As Section 2.1 emphasizes, challenge is a critical component of fun in many videogames. For this reason, our model draws an explicit relationship between challenge (as experienced in the form of repeating rhythm groups) and fun. Our formulation of this relationship is informed by the nature of rhythm groups, which can be observed in real-world level designs. We also take into account the Yerkes-Dodson law [61] and its relevance to videogames, as documented by Piselli et al. [38].

3.1 Model Scope

A model, by necessity, entails simplification. Because our model expresses the relationship between challenge dynamics and fun in videogames, we must clearly define what is meant by the term 'fun,' what types of challenge are implied, and what types of videogames best exhibit this kind of challenge.

3.1.1 Fun

Certainly, the notion of fun is a nebulous concept, and the specification of the model's characterization of quality requires particular attention. We emphasize that the purpose of our rhythm-group model is to serve as a fitness function in a generative process, and,

therefore, to evaluate the relative quality of generated levels. Because our goal is not primarily to model a psychological state of fun, we use the term 'fun' in a more pragmatic sense: as a measure of the quality of a level's design. According to this sense of the term, validation for the model does not come from subjective studies with human participants but, rather, from observations of the the game industry's classic, enduring examples of good level design. We do not entirely distance ourselves from the notion of fun as a pleasurable mental state; certainly, a level of high quality is essentially a level which gives pleasure to the player. However, for our purposes, we assume that the properties that elicit this state of pleasure are sufficiently manifest in the level designs themselves and that we can, therefore, understand important qualities of challenge-based fun by restricting our analysis to the level designs alone. In other words, the model is considered successful insofar as it is able to attribute high fitness values to levels which exhibit structures characteristic of those found in well designed levels.

3.1.2 Challenge-Based Games

Our model of fun applies to challenge-based action games. These are games in which the predominant form of pleasure does not arise from exploration, online social interaction, logical puzzle solving, or narrative, but rather from reflex-based tests of skill. Indeed, as Section 2.1 emphasizes, there are so many aspects to the concept of fun in general that it becomes necessary to restrict our focus to a single area; we claim that reflex-based challenge is more amenable to formal analysis than, for example, narrative or aesthetic pleasure.

To define our scope even more precisely, we are interested in games in which the challenge is delivered primarily through the level design. This requirement includes genres such as platformers (e.g. *Mario* [32]), action-adventure games (e.g. *Zelda* [31]), and first-person shooters (e.g. *Half-Life* [59]). This requirement notably excludes genres such as fighting games (e.g. *Street Fighter* [35]) and sports games (e.g. *NHL 11* [13]). In these games, the nature of the challenge, and ultimately the fun experienced by the players, is primarily a function either of the skill of the opponents (artificial or human) or of the rules and controls governing the game mechanics. The levels of such games serve merely as an aesthetic backdrop to frame the game and do not serve as a promising target for automatic generation. This distinction justifies the use of our model as a *direct* fitness function for level evaluation, a notion which Togelius et al. [55] defines in contrast to *simulation-based* fitness functions, which depend on a dynamic observation of actual game-play to assess quality.

3.1.3 Rhythm Groups

Our usage of the notion of rhythm groups differs slightly from that described by Smith et al [47]. Whereas Smith et al. define rhythm groups in terms of timed sequences of button presses, analogous to rhythmic beats in musical compositions, we emphasize the more general notion of oscillating challenge over time. Because the notion of rhythm groups is not popularly used in the industry, we can use the term to refer to the level structures that our model identifies instead of appealing to an authoritative external definition. However, it is not a completely arbitrary concept, as we have a precise usage in mind: rhythm groups correspond to recognizable moment-to-moment fluctuations in challenge on the scale of seconds. It is this scale that best serves the purposes of identifying the quality of single levels.

3.2 Model Design

The model's purpose is to estimate the amount of fun provided by a particular level, based on that level's configuration of challenge in the form of rhythm groups. Our model attributes high values of fun to levels containing rhythm groups that resemble those found in commercial games, and its ultimate purpose is to evaluate and guide the output of our generative system. For the model to be deemed useful, therefore, it must reward levels which exhibit a number of important properties that are noticeably present in real-world levels.

3.2.1 Level Example Set

Twenty-eight levels taken from *Super Mario Bros.* were used to inform the design of the model. These levels were chosen according to the degree to which their design contributed to the ultimate challenge dynamics of the game. This criterion excluded "boss levels," as the difficulty of these levels is primarily determined by the patterns of movement of the final enemy. Some other levels were excluded, such as ones containing a Cloud Koopa enemy, who follows the player, hurling new enemies for the duration of the level. Underwater levels were also excluded due to their significant differences from the rest of the game environments. Excerpts from four of the 28 chosen levels are shown in Figure 3.1.

The levels were reconstructed by Ian Albert from recorded screen captures of play sessions, which are made available on his website [1]. We convert these screen captures into

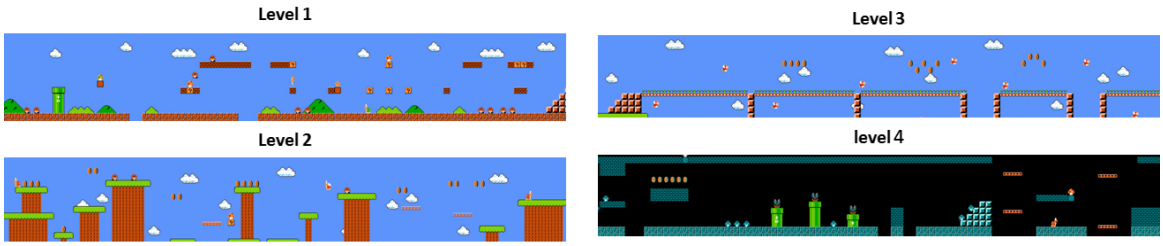


Figure 3.1: Segments of four of the 28 levels used to inform the design of the model of fun.

a format more amenable to analysis though computer vision techniques; the basic sprites corresponding to the various types of blocks and enemies are identified, and the location and distribution of each object type is found through template matching. We determine enemy locations from their sprite coordinates and find hole locations by detecting gaps in the blocks located at the bottom of the level. This information is converted into a time series representation of the "challenge events," a signal that is zero everywhere, except for unit impulses at the places where enemies or holes are located. The particular distribution of these challenge events is what we hypothesize as predominantly affecting the amount of fun had by players. The resulting time series are shown in Figure 3.2.

From examining the 28 test cases, we draw the following generalizations, which we will later use as evaluative criteria for our model:

1. Too much continuous difficulty is undesirable. Though each rhythm group presents the player with a high degree of challenge, it is not so great as to cause frustration and a reduced sense of pleasure for the player.

2. Rhythm groups are not necessarily strictly periodic. They certainly exhibit a cyclical pattern; however, there is no evidence that there is a predominant frequency to the amplitude of challenge over time (see Figure 3.3). Levels with variously spaced rhythm groups should not be penalized.

3. The model should account for players of different skill levels. The model should not be fragile in the sense that it applies only to a single, idealized player but should rather be adaptable to a wide variety of players through the manipulation of a few, semantically meaningful parameters.

4. Rhythm groups should conform to a reasonable scale on the order of seconds; it should not be possible for a generative system to exploit the model with obvious degenerate

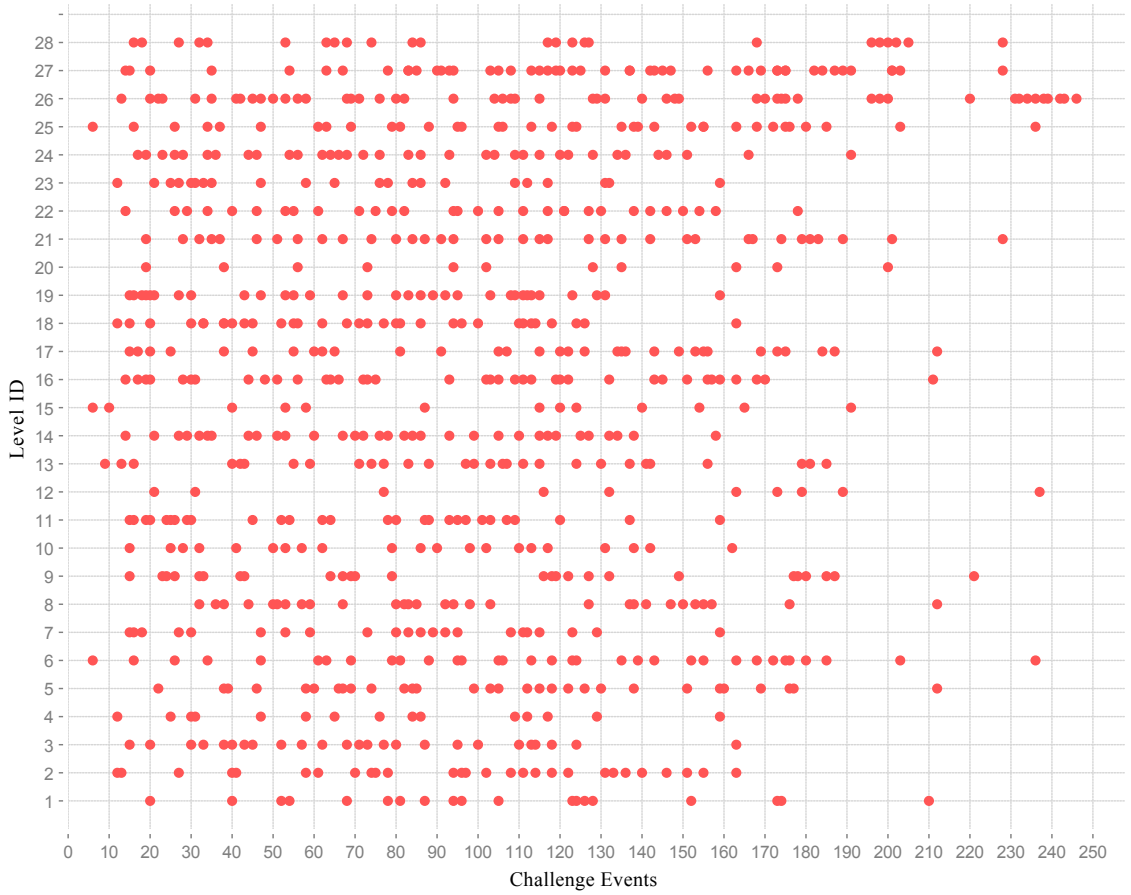


Figure 3.2: Depiction of the challenge events of the 28 levels used to inform the design of the model of fun. Horizontal scale is in 16×16 block units. Note the visible clustering of challenge events into rhythm groups.

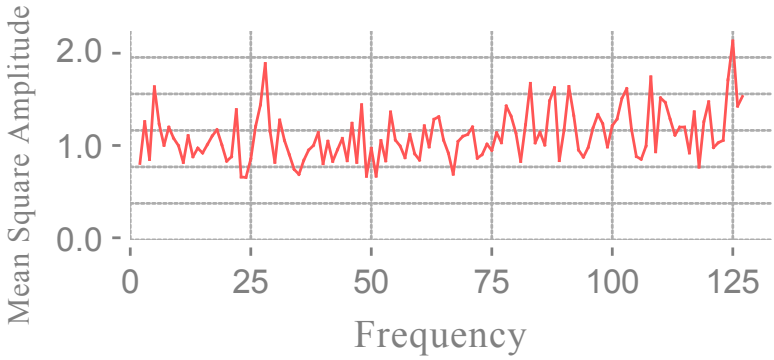


Figure 3.3: Power spectrum of challenge dynamics of the 28 *Mario* levels taken as a whole, demonstrating the frequency distribution of challenge events. Levels are interpreted as time series 256 units long with holes and enemies (challenging segments) representing unit impulses of unit amplitude. Vertical scale is 10^{-3}. Note that the distribution of challenge does not exhibit a dominant frequency, but instead resembles white noise.

constructions, such as levels containing a single rhythm group lasting for the entire duration of the level. The model is most useful for the purposes of level generation if it is sensitive on a small scale and is able to identify even minute improvements in a level's layout.

3.3 Model Formalization

3.3.1 Challenge

Because rhythm groups are defined in terms of challenge dynamics, our model presupposes the existence of a suitable technique to measure the change of challenge over time. It is outside the scope of this work to address the notion of challenge generally; thus, we must assume that the model is provided with a challenge metric $c(t)$, which returns, for a given level, the degree of challenge at the time t. Certainly, the manner in which this value is calculated will vary depending on the game context. For example, with *Mario* we use a challenge function that identifies a portion of a level with any given value of t and associates difficulty values based on the design of the level at that given point. In particular, a large gap with a small margin of error for mistakes will be attributed a relatively high value of difficulty, whereas a large, straight segment with no enemies will be given a challenge value of zero.

The construction of the challenge metric entails certain simplifications. For one, we must treat challenge as a single dimensional value. We also assume that the challenge metric is always non-negative. While the particular values do not matter (all we are concerned with is relative ordering), we take zero to represent the lowest possible challenge experienced in the game.

We also treat challenge as a value that can be sampled at a single point t. Chapters 5 and 6 provide example formulations for $c(t)$ in the context of two different games and demonstrate the modeling of challenge as instantaneous impulses, that is, formulating $c(t)$ as a sum of Dirac delta functions (unit impulses), as illustrated in Figure 3.4. It might seem that challenge is not actually experienced in terms of discrete impulses but is rather a dynamic quality that must be observed over a duration of time, much like tonal pitch in an audio sample. However, as Section 3.3.2 will demonstrate, challenge is, indeed, subjected to integration over the time window, which is determined by the size of the enclosing rhythm group. For this reason, regardless of whether the challenge function is formulated as a sum of discrete impulses or as a continuously varying value, the function contributes to the model in the same manner.

Finally, we take the value of challenge to represent the difficulty of a certain level segment absolutely, that is, irrespective of player skill. Thus, if we were to suggest a possible unit of measure for the challenge metric, these units would be constant for everyone, not relative to a particular personal experience of that level. Again, this does not prove to be a problem as the model provides threshold parameters which account for the fact that skilled players would be capable of enjoying higher degrees of challenge than players with less skill. In other words, skilled players could be exposed to a higher amount of these hypothetical "challenge units" before becoming frustrated.

Finally, we emphasized that because $c(t)$ is ultimately used as a component of an automatic fitness function, it needs to be calculated without any human input; it must be possible to find a reasonable estimate of a level's configuration of difficulty over time simply by analyzing the level's layout. This is the primary reason why we restrict our discussion to challenge-based games in which the difficulty is mostly a function of the level design.

3.3.2 Modeling Fun

The determination of a level's quality consists of a two-pass process. First, the level is partitioned into a set of n rhythm groups with boundaries located at times t_0, t_1, ...,

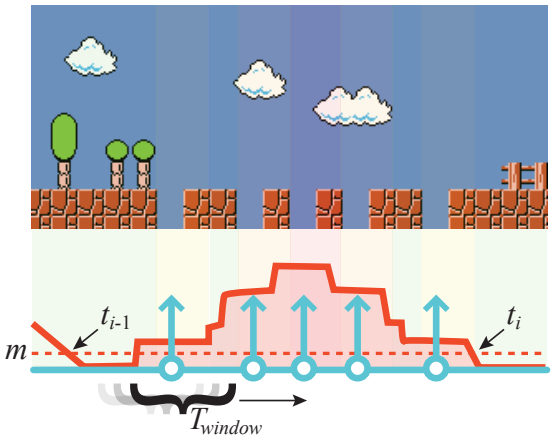


Figure 3.4: Illustration of rhythm group i in the context of *Mario*. Vertical arrows represent the challenging events (holes) as unit impulses, and the curve represents the amount of accumulated challenge in the time window. Rhythm-group boundaries are located at points t_{i-1} and t_i, because the windowed challenge temporarily decreases below the threshold m. The accumulated challenge in the entire rhythm group, c_i, corresponds to the integration of the impulses located between boundaries t_{i-1} and t_i .

t_n. The identification of rhythm-group boundaries is governed by a greedy algorithm that identifies periods of sufficiently low challenge, which are identified as periods of relaxation. A window of size T_{window} is shifted along the challenge function, and positions the boundaries at points where the total amount of challenge in the window is less than the threshold m. More precisely, boundaries are located at positions t where $\int_{t-T_{window}}^{t} c(t)\,dt \leq m$. After a boundary is associated at a given point in time, the window does not place any more boundary points until after it has witnessed a period where the challenge temporarily exceeds m. Otherwise, extended periods of low challenge would be identified with a dense interval of infinitely many rhythm groups. Put more simply, the greedy process only identifies a new period of relaxation until after an intervening period of challenge has elapsed. A rhythm-group boundary is always placed at the beginning and end points of a level, and because the process is greedily run from the beginning to the end, it produces a unique segmentation.

With this segmentation in place, it is possible to identify the level with a fitness value. A level is rewarded for each rhythm group that contains the appropriate amount of total accumulated challenge (which we refer to as "anxiety") as specified by the upper threshold M. Formally, if the amount of anxiety contained in rhythm group i is given by $c_i = \int_{t_{i-1}}^{t_i} c(t)\,dt$, then the amount of fun, f, attributed to the level as a whole is defined by (3.1).

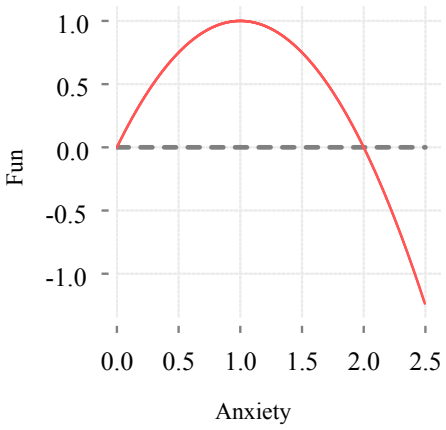


Figure 3.5: Fun, f, as a response to increasing anxiety (accumulated challenge), c_i, when $M = 1.0$, in the context of a single rhythm group. The response is defined by the function $f = \frac{2c_i}{M} - \frac{c_i^2}{M^2}$.

$$f = \sum_{i=1}^{n} \frac{2c_i}{M} - \frac{c_i^2}{M^2} \tag{3.1}$$

3.4 Model Characteristics

The numerical response of the model is demonstrated in Figure 3.5, which illustrates the amount of fun in a particular rhythm group as a function of the accumulated challenge in that rhythm group. Recall that accumulated challenge—that is, challenge integrated over a period of time—is referred to as "anxiety." In other words, where $c(t)$ represents the amount of challenge present at the instantaneous point t, c_i represents the total amount of challenge integrated over the duration of rhythm group i, which constitutes a quantity of anxiety. The rhythm group attains its maximal fun potential when the amount of anxiety present is exactly M. The fun provided by a rhythm group decreases if the amount of anxiety experienced in that group is greater or lesser than this critical point. This function is evaluated independently for each rhythm group, and the fun for the entire level is the sum of each independent evaluation.

We define the formula depicted in Figure 3.5 in order to mimic the famous "inverted U" shape described by the Yerkes-Dodson law [61], which essentially states that task performance is optimal if arousal is neither too high nor too low. Piselli et al. [38] have shown that

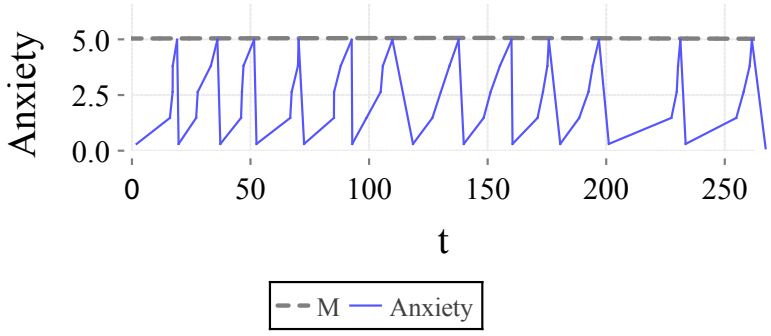
this phenomenon is equally applicable to the context of videogames, even when considering pleasure as a function of difficulty instead of task performance. Our model, then, idealizes this phenomenon within a computational setting, and applies it at the scale of individual rhythm groups.

To illustrate the effect of using this model to guide a generative system, we employ the model as a fitness function in a genetic algorithm that evolves challenge time series directly, with no reference to an actual game context. The genotype is a variable-length list of challenge locations, which corresponds exactly to our representation of the 28 levels drawn from *Mario*. We use typical GA settings, with crossover at 0.9 and mutation at 0.05, and stop the evolutionary run when progress has stalled for 10 generations. The results are shown in Figure 3.6.

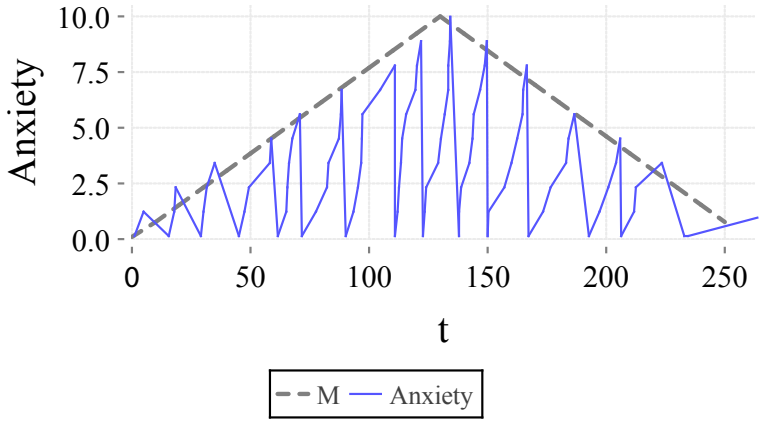
Our model satisfies the four motivating properties outlined in Section 3.2. First, it is evident that the model penalizes levels with excessively high difficulty values; the reward for a given rhythm group attains its maximum at the value M and decreases quickly after that threshold has been surpassed. Second, there is nothing in the formulation that encourages strict periodicity; fitness is rewarded solely based on the amount of challenge present in a rhythm group and is not predicated on the rhythm group conforming to any specific width.

As well, the model can easily and intuitively be adjusted to account for players of different skill levels. The parameter M corresponds to the skill of the player and can be increased or decreased to create levels of higher or lower difficulty. As Figure 3.6(b) shows, this parameter can even be adjusted over the course of a single level, providing the designer with the ability to control the overall arc of challenge, in this case creating a level that has a very challenging mid-point with easier portions at the beginning and end. It would similarly be possible to adjust m dynamically in order to increase or decrease the lower bound of anxiety required to trigger rhythm-group boundaries.

Finally, the above model does not lead to degenerate cases, as rhythm groups that are too long impose a low upper bound on the total amount of fun that can be attributed to a level. If a level were to be identified as consisting of a single rhythm group, then the amount of fun attributed to that level would be, at most, 1. Any level with more numerous (and smaller) rhythm groups will certainly be able to exceed that value and be favoured for selection. On the other hand, rhythm groups must be at least a width of T_{window}, which places a lower bound on their size. In this way, these two bounds ensure that rhythm groups exist at a scale that can allow a level design to be analyzed at a meaningful resolution—on



(a) Constant M



(b) Varying M

Figure 3.6: Challenge time series with high fitness, as induced by the model with $T_{window} = 10$, and $m = 1$. The curve depicts accumulated challenge in each rhythm group (c_i). Notice that the anxiety in each rhythm group attains M.

the order of seconds, not minutes.

3.5 Learning Parameters

Provided that a suitable challenge metric $c(t)$ is defined and that values are specified for the parameters M, m, and T_{window}, the model is able to estimate a level's entertainment value. However, it is possible to perform the reverse operation, beginning with a set of levels of a known entertainment value and working backward to infer the specific model parameters that reproduce the observed values. This task is an instance of an expectation maximization problem, with the goal of finding model parameters that account for an observed set of example levels. By following this procedure, it becomes possible to mimic the particular challenge characteristics of an existing game; given a set of levels that are considered to be well designed, the model can be trained to reward levels with similar challenge configurations. This could be an effective way to expand the content of a game, as any automatically-generated levels would exhibit the same difficulty patterns as the human-designed levels.

The inference of the model parameters also serves as a means of validating the model itself. If certain parameters allow the rhythm-group model to successfully distinguish between well-designed and poorly-designed levels, it stands to reason that the model is sensitive to relevant characteristics of high quality level design. Indeed, this validation is of critical importance when asserting the usefulness of analyzing levels in terms of rhythm groups. We do not currently rely on qualitative, subjective evaluations of fun (such as questionnaires) to evaluate the model; instead, we can assess the model's effectiveness by treating it as a classifier of existing, real world levels, judging its performance in the same way many other machine learning techniques are judged. Furthermore, this approach provides an avenue of testing the model independently of any particular generative context. Evaluating the output of a generative system that is based on the rhythm-group model does not necessarily demonstrate that the model, itself, is responsible for the quality of that output; it could be the case that the generated levels are fun only because of some felicitous property of the generative system. However, in evaluating the model in isolation, we can more confidently defend the use of rhythm groups as a meaningful analytical tool.

3.5.1 Classification Results

It proves to be the case that the model is, indeed, able to successfully distinguish between the challenge time series of the 28 levels taken from the original *Super Mario Bros.* game, which are considered to be examples of good level design, and 30 time series that were crafted to represent examples of poor level design. We constructed the 30 negative examples to represent properties that would be obviously undesirable in well-designed levels, arguing that the model would certainly need to be able to identify these levels as poorly designed before it could be seen as effective. In that sense, this experiment establishes a base-line functionality and serves as a "sanity check" of the model's usefulness.

Because we have seen that actual levels have no regular periodicity, we have generated negative examples that do exhibit a regular, periodic structure. We have 14 negative examples intended to represent levels that are clearly too difficult. Some contain challenge impulses located 1 to 3 units apart for the entire duration of the level, while others contain bursts of 20 contiguous challenge impulses located between 20 and 30 units apart. Similarly, we have 16 examples of levels that would be too easy, with single challenge impulses separated by 20 to 30 units of space. Illustrations of these negative examples are provided in Figure A.1 in Appendix B.

The model can be converted to a classifier through the addition of an extra parameter, θ, which represents the threshold of fun above which a level is considered to belong to the class of well-designed levels. For the sake of convenience, we train the classifier using the evolutionary system we have in place. We treat model parameters as individuals in a population and define a fixed-length genotype defined by the tuple $(T_{window}, M, m, \theta)$. The fitness of a set of parameters is given by the proportion of correct classifications of the training data. Optimal values were routinely found within the span of a few generations, and evolution was stopped if there was no fitness improvement after 5 generations.

The experiment was conducted as a 10-fold stratified cross validation by randomly partitioning the 58 training examples (consisting of the 28 real *Mario* levels and the 30 hand-designed negative examples) into ten groups of five or six examples each, with roughly three positive and three negative examples per group. The model was then trained on each of the ten different groups, which were formed by removing one of the sets for validation purposes (so that the same data points were never used for both training and validation). The model was quite successful at distinguishing our real levels from the poorly designed levels; with

Parameter	Mean	Variance
T_{window}	9.5	6.6
M	7.3	3.5
m	1.5	0.2

Table 3.1: Mean and Variance for the 10 optimal parameter settings found through cross validation.

only two misclassified examples, it achieved a 97 percent successful classification rate. More details about the specific results of each of the ten training runs is provided in Table A.1 in Appendix B.

Also encouraging is the fact that the optimal parameters conformed to their intuitive roles in the function. As shown by their mean values in Table 3.1, rhythm groups corresponded to periods containing an average of 7 challenge impulses, and inspection of the 28 example levels reveals visible clusters of challenge events containing roughly that many items. It is reasonable that a period of about ten blocks with only a single hole or enemy (according to T_{window} and m) would constitute a rhythm-group boundary. In other words, it is a reassuring result that the parameters which correspond with an intuitive, visual inspection of actual *Mario* levels are precisely the parameters that lead to successful automatic classification under the proposed model.

3.6 Summary

The model outlined in this chapter draws an explicit relationship between challenge dynamics and fun in the context of challenge-based videogames. Inspired by several levels drawn from the classic game *Super Mario Bros.*, the model satisfies the four design goals we outline in Section 3.2.1: it penalizes levels with too much continuous difficulty; it does not require rhythm groups to be strictly periodic; it can be adapted to players of differing skill levels; and it identifies rhythm groups on a scale of seconds. We have confirmed these observations and validated the model by testing whether or not it is able to successfully distinguish between actual *Super Mario* levels and levels which violate our design goals. With a 97 percent successful rate of classification, it is clear that the model is, indeed, sensitive to relevant information regarding challenge dynamics. This learning process also demonstrates

a method to find effective parameter settings that correspond to real-world data.

Not only does the model satisfy our design criteria, but it does so in a justified manner. Through a literal interpretation of the Yerkes-Dodson law, rhythm groups are deemed to be maximally fun when they are neither too easy nor too challenging (according to precise operational definitions of these terms). This formulation, in making explicit the relationship between challenge and fun, allows the model to provide semantically meaningful parameters that have intuitive relations to real world phenomenon. For example, the parameters M and m represent the upper and lower bounds of a player's skill and can be manually adjusted to reward levels of varied degrees of difficulty: levels with relatively high degrees of difficulty will be associated with high values of fun if M is set to a large value, whereas levels with relatively low degrees of difficulty will be identified as fun should M be set to a small value. Such parameters provide game designers with a high level of control over the generative process that is not afforded by bottom-up techniques such as fractal-based generators or grammars. By varying M over time, designers can exert predictable, local control over the design of a level, creating, for example, levels that exhibit a dramatic arc with an easy beginning and difficult end. This differs from generative grammars where the alteration of a single threshold parameter would have global effects and alter the design of the entire output.

Finally, the model is well suited to serve as an automatic fitness function. A real-valued measure of level quality can be produced through the analysis of a level's challenge dynamics, and it is not necessary for a human to play and subjectively evaluate a level's quality. Since we can compute the fitness of the thousands of potential designs in this manner, the evolutionary run can consider many more variations than would be possible with interactive evaluation. Because our model is real-valued, it can also make fine distinctions between relative degrees of fun. This property is essential for evolutionary search; because initial populations are likely to contain only designs that are not fun, it is critical to be able to distinguish between even low degrees of fun in order to guide the population toward beneficial mutations. Because rhythm groups are identified on a scale of seconds, the model is sensitive to even small improvements in a level's structure.

Chapter 4

Implementation

The generative process is ultimately a search through the space of possible designs. The system attempts to find a particular level that demonstrates a good configuration of rhythm groups and that possesses, therefore, a high fun value. This section goes into more detail regarding the particular techniques used to successfully traverse this space.

At the core of the approach lies a genetic algorithm (GA), for which each potential level design is represented by a genetic encoding. We extend the basic algorithm with new features that help to overcome some difficulties associated with evolutionary search. Constraint satisfaction methods are employed to form a hybrid system that effectively optimizes the value of fun for levels while simultaneously observing the strict constraints inherent to level design.

The system is written in the language Clojure [19], which is a Lisp dialect that runs on the Java Virtual Machine. Clojure is a purely functional language that provides a comprehensive set of immutable data structures and concurrency constraints. These features allow our system to easily leverage multiple cores for efficient parallel computation. As well, because it is a highly interactive and dynamic language, the system could be rapidly modified to reflect changes to our model and generative approach during development. Finally, because the system operates on the Java platform, we are able to interface with Java libraries. The core of the system was drawn from earlier work on the evolution of match-3 puzzle games, and was extended in the implementation of our current system. Including the code needed to apply the system to the various games, the project as a whole currently comprises approximately 4000 lines of code.

4.1 Problem Domain

Level design remains a challenging AI search problem for two primary reasons. First, it is a task characterized by high dimensionality; a single level design in our system contains hundreds of degrees of freedom. Genetic algorithms are an effective tool when approaching this kind of problem, as they are well suited to such high-dimensional search spaces. However, level design is also a highly constrained task. Level elements must be arranged in such a way as to ensure that the player is able to traverse the level. For example, if a platform in *Super Mario Bros.* was placed too far from a ledge for a player to reach, the entire level would be rendered unplayable. An objective function would typically associate completely broken levels such as this with a fitness value of zero. Because a small change in the positioning of a single element can drastically change the objective quality of a level, the domain is deemed to have a highly discontinuous fitness landscape, suggesting that the problem is poorly suited to an evolutionary approach. In cases where the constraints between solution elements are critical, constraint satisfaction methods are more appropriate.

The generative approach presented in this thesis is constructed to address both concerns simultaneously and is an example of a hybrid constraint solver and evolutionary system [9]. Such systems strive to observe the constraints of the problem domain while exploring a high dimensional search space in order to maximize an objective fitness function.

4.2 Genetic Representation

The genetic representation of a level is a variable-sized, unordered set of *design elements* (DEs). Design elements are atomic units that combine to form a game level. Intuitively, DEs represent the components a human level designer would arrange when manually constructing a level for a game. For example, a single enemy in *Mario* is represented as a DE. A given game will include a number of different types of DEs, which together express the breadth of elements available to the game designer. More detailed examples of constructions of DEs for specific games are offered in Chapters 5 and 6.

Each type of DE is defined by a number of parameters and is essentially a tuple containing floats, integers, and Booleans that represent the characteristics of that DE. For example, in an adventure game, an enemy DE might have two dimensions representing its horizontal and vertical position, one dimension representing its strength, and a Boolean dimension

determining if it is armed or unarmed.

4.2.1 Genetic Operators

Mutation is accomplished with respect to a single DE. A single mutation operation can either be the addition of a new random DE to the genotype, the deletion of a DE from the genotype, or the modification of one of a DE's constituent property dimensions. A new DE can be created by selecting from one of the game's basic types of DE and setting its dimensions to random values in their respective domains. Mutation of a DE is achieved by selecting a new value for a random parameter. If the parameter is a real-valued number or an integer, the parameter is perturbed to a degree defined by a Gaussian distribution. If it is an unordered categorical variable (including Boolean parameters), it is set to a new allowable value with uniform probability. Each DE dimension can also be associated with a scaling parameter that affects both the scale of the Gaussian distribution and the variance of the mutation which is applied to a particular dimension.

The crossover operator is similar to variable-point crossover but modified slightly to be compatible with our representation. Standard variable-point crossover is achieved by picking a random cut point in the two parent genotypes and swapping two halves of each split parent genotype to create two new offspring. Our genotype representation consists of an unordered set of DEs, and typical crossover operators are defined in terms of ordered, linear genotypes, so standard variable-point crossover cannot be applied directly. However, because our DEs represent substructures with spatial position, we can impose a linear order by sorting along a spatial dimension. This approach is applicable to any n dimensional space; every crossover involves picking a dimension at random, sorting by that dimension, and behaving exactly as a variable-point crossover on the now-linear representation. For example, in a two-dimensional context, the parents will be split by a random horizontal or vertical plane, and the offspring will be formed by taking all the DEs that lie to one side of the plane from the first parent, as well as all the DEs that lie on the other side of the plane from the other parent.

This approach serves to draw together within the genotype DEs which represent level structures that are in close proximity, providing the property known as gene linkage [17]. An important aspect of any genetic representation is the strength of the gene linkage, which determines the efficacy of the crossover operation in preserving useful modular sub-structures. In the worst case, when the DEs have an arbitrary ordering, the genetic algorithm degrades

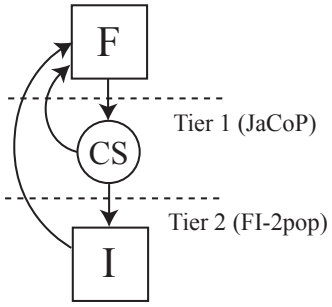


Figure 4.1: Overview of constraint satisfaction system. Individuals who violate design constraints are first passed to the Tier 1 subsystem (CS), powered by the JaCoP constraint solver. Repaired individuals are moved back into the feasible population (F). Individuals that could not be repaired are moved to the infeasible population (I) in the Tier 2 subsystem, which is driven by the FI-2pop GA.

into regular hill-climbing (albeit with large, random, and disruptive changes interspersed with smaller mutations). Strong gene linkage, however, is what enables genetic algorithms to preserve high-fitness sub-structures throughout a population, which would be otherwise destroyed through small-step mutations.

4.3 Constraint System

Constraint satisfaction (CS) methods are added to the typical genetic algorithm structure in order to address the challenges of a highly constrained solution space with a discontinuous fitness landscape. We use constraint satisfaction to repair the genotypes that are subjected to breaking changes. We use two distinct forms of CS, which address two particularly relevant forms of level design constraints. Constraints can be formulated as local, spatial relations such as "the object X must not overlap the object Y." These constraints can be solved with the "Tier 1" constraint satisfaction system, which immediately alters the genetic representation to directly satisfy the constraints. Not all constraints can be easily expressed in terms of local, spatial relations, however. For example, the constraint "there must be an unblocked path between the points A and B" cannot be easily expressed as a geometric constraint between two elements. These more complex constraints are handed by the "Tier 2" constraint system. An overview of the constraint satisfaction architecture is depicted in Figure 4.1.

4.3.1 Tier 1 System

Tier 1 is an example of a typical constraint solving algorithm that employs variable selection, domain pruning, and backtracking. We use, specifically, the JaCoP open source Java constraint solving library [26] as the foundation of our approach and modify it to better suit our use of it as a reparation step in a genetic algorithm. JaCoP is particularly useful as it features a geometric constraints module which allows many constraints typical of spatial arrangements to be straightforwardly expressed and efficiently solved.

Because of JaCoP's role as a reparation step in a larger process, we are concerned with more than simply finding a set of values that satisfy the problem constraints. We want to ensure that the reparation process modifies an existing genotype as little as possible in its attempt to provide a viable solution. The benefits of this are twofold. First, because the genetic algorithm uses the rhythm-group model to evaluate designs, the more we alter a given level design to satisfy game-specific constraints, the more likely we are to disrupt the rhythm-group structure and reduce the effectiveness of the model in producing fun levels. A second benefit to altering the genotype as little as possible is the possibility of implementing mixed-initiative design. We can use the same machinery that minimizes genotype modifications to ensure that the system respects the designer's adjustments to the level and alters the level in such a way as to minimize disruptions to content made by human design collaborators.

These requirements are not a part of the typical constraint problem formulation, which seeks only to set variables to values within acceptable ranges. In our formulation, all variables are already populated with values because the input is an individual from a genetic algorithm. For our purposes, we wish to alter the variables as little as possible to satisfy the constraints. To put this more precisely, we define a cost function for a possible solution set shown in (4.1), which is a summation over every dimension of every DE (denoted by *DED*) with $\textit{altered?}_i$ returning 1 if the DE i has been altered, and 0 otherwise, $\textit{altered-degree}_i$ returning the difference between the altered dimension value and value of the original, and α and β as scaling parameters.

$$\textit{Cost} = \sum_{i \in DED} \alpha \cdot \textit{altered?}_i + \beta \cdot \textit{altered-degree}_i \tag{4.1}$$

Our constraint satisfaction problem can then be framed as a search for values for the DE

dimensions that minimize (4.1). This is achieved by extending the JaCoP constraint satisfaction library with alternate variable selection and value selection processes. Constraint solving requires picking both a variable to alter and a new value for that variable, and our approach is to prioritize the choice of values so as to reduce the potential negative impact. In other words, if the original value of a certain DE dimension, as set by the GA, is 3, the values are chosen in the order of increasing distance, e.g.: $[4, 2, 5, 1, 6, 0, 7, -1, \ldots]$. We do not claim that this approach always finds the absolute global minimum of (4.1), but it is more effective at approaching this goal than an arbitrary value assignment.

Every individual that can be successfully repaired through the Tier 1 system is fixed and placed back into the population.

4.3.2 Tier 2 System

The Tier 2 System handles individuals that cannot be repaired through the Tier 1 process, and it can satisfy certain constraints which cannot be easily expressed using the primitives provided by the JaCoP system. The subsystem, which is described in earlier work [50], is modeled after the Feasible/Infeasible 2-Population Genetic Algorithm (FI-2pop), developed by Kimbrough et al. [24]. The FI-2pop consists of two populations which are evolved in parallel, one labeled the "feasible population," which contains all the individuals that satisfy the constraints of the problem domain, while the other is referred to as the "infeasible population," which contains those individuals that do not satisfy the constraints. In our case, the feasible population contains all the levels that satisfy the constraints of the game in question, as well as those individuals that can be repaired by the Tier 1 constraint solver so that they do not violate any constraints. The individuals which cannot be repaired, or which violate constraints that cannot be expressed in the terms of the Tier 1 subsystem, are placed in the infeasible population.

Whereas the feasible population is evolved according to our primary fitness function, that is, by the rhythm-group model of fun, the infeasible population is evolved according to a fitness function which seeks only to satisfy the still-violated constraints. This is done with a measurement of the degree to which a given level violates the set of constraints. By minimizing this function, levels ultimately reach a state where they violate no constraints, at which point they can be moved back into the feasible population. Because Tier 2 constraints are enforced through a fitness function, they can express any arbitrary, global constraint on a level design; they do not need to be limited to spatial relationships between individual

level elements. An example of a global property that is difficult to express in terms of local constraints is connectivity—ensuring there is a traversable path from the beginning to the end of a level.

4.4 Island Model Extension

Although the FI-2pop is an effective model for performing evolutionary searches in highly constrained domains, it lacks some of the basic extensions that have been used in conjunction with typical genetic algorithms. Because of the FI-2pop's similarity to basic GAs, it is possible to describe an island model that closely parallels the methods outlined in the literature. In fact, the definition and implementation of this extension are contributions of our work. This extra functionality improves the performance characteristics of the evolutionary search by allowing for greater parallelism and enables more difficult design problems to be approached than would otherwise be possible.

Population diversity is essential for sufficient exploration of a design space in an evolutionary search, yet basic genetic algorithms tend to lose diversity very quickly. Without any pressure to discourage uniformity, the first local maximum found will quickly dominate the population. One of the most common ways to address this is to adopt the island model of evolutionary algorithms. Essentially, several different evolutionary runs are executed in parallel, ensuring the current fitness leader does not lead to the extinction of all other developing solutions. In addition, the best individuals from each island are occasionally migrated to neighbouring islands, allowing for cross fertilizations to aid in escaping local maxima. The frequency of these migrations can be adjusted; very frequent migrations reduce the island model to one large, basic GA, which quickly loses diversity, and very infrequent migrations amount to merely running independent GAs in isolation. Migrations are typically set to happen every 50 generations, which is the value found to generally provide the best trade-off between resisting population homogeneity and cross-fertilization.

The FI-2pop already resembles the island model, with one population, F, containing feasible individuals, and the other, I, representing infeasible. We define an island configuration in FI-2pop to be n pairs of feasible and infeasible populations, F_i, I_i, where $i \in [1 \dots n]$, each containing p individuals. Individuals from feasible populations are migrated to other feasible populations, and individuals from infeasible populations migrate to other infeasible

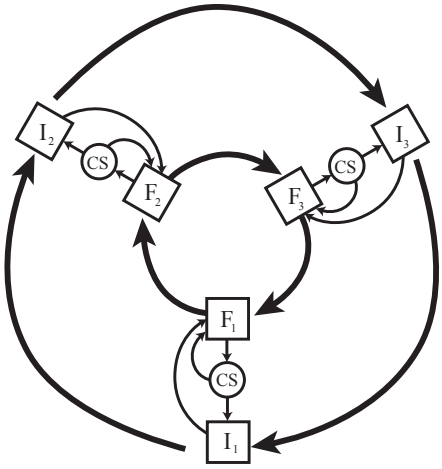


Figure 4.2: Generative system arranged in a 3-island ring topology. Individuals in the feasible populations F_1, F_2 and F_3 migrate clockwise to their nearest neighbour. Infeasible populations migrate similarly.

populations. These populations are arranged according to a ring topology, where $\lfloor \frac{p}{2} \rfloor$ individuals are copied to their neighbouring populations during migration, replacing the $\lfloor \frac{p}{2} \rfloor$ worst individuals of the destination island. This arrangement is depicted in Figure 4.2.

The island model provides a straightforward and efficient way to split an evolutionary search across multiple threads for more effective computation on a parallel processor, as each island naturally corresponds to a processor thread. As Kimbrough et al. explain, however, the two populations of the FI-2pop are evolved synchronously, with each new feasible generation corresponding to a new infeasible generation. Because the amount of time to produce a generation can vary greatly, due in large part to the constraint solving operation in Tier 1, we find that evolving the feasible and infeasible populations asynchronously results in less idle time between generations. As well, since global synchronization no longer exists between populations, the migrations are triggered by a global counter. After $50n$ generations have been conducted for all n islands together, migrations will occur.

4.5 Summary

This architecture maintains all the advantages of the top-down approach. It allows the level design criteria to be described declaratively, irrespective of the actual generative implementation. Procedural generative processes are restricted to the reification of the DEs (through the genotype to phenotype mapping process) and our architecture provides a clear separation between the mechanics of the level creation and the evaluation process. Furthermore, this system is compatible with the goals of mixed-initiative design; the DEs could be provided by a designer through their usual level design editor and given special status in the system. The GA treats these DEs as fixed and immutable and does not alter them under mutation or crossover. Likewise, the constraint satisfaction system gives a higher weighting to the variables corresponding to the DEs that are provided by the human designer when altering their values, preferring to mutate automatically-generated DEs provided by the GA than to change the DEs specifically placed by the human. Essentially, this amounts to a system that works around and with a human designer to "fill in the blanks," upsetting as little as possible both the guidance of the rhythm-group model and, more importantly, the designs provided by the human.

Chapter 5

Evolving Mario Levels

In this chapter, we outline a concrete application of the model and generative framework in the context of an actual game, namely the 2D platformer *Super Mario Bros.* First, we describe a challenge function for this game in order to apply our rhythm-group model. We present two ways to model this design task in terms of DEs and constraints. The first formulation is simpler than the second and serves to demonstrate both the simplicity of specification from the designer's perspective and also the system's ability to produce playable levels with a minimal amount of game-specific knowledge. This supports our claim of generality, since generality is partially dependent on avoiding as many game-specific details as possible. The goal of the second, more complex formulation is to demonstrate the system's ability to generate levels of comparable quality, in terms of design and entertainment value, to those produced by a human designer.

The system produces levels that are directly playable. The game we use is an open source clone of *Super Mario Bros.* called *Infinite Mario.* It is written by Markus Persson in the Java language and is currently used in several videogame research problems, including the Mario AI Championship, where it functions as a platform for testing the performance of various AI character controllers, learning agents, and generative systems, such as our own. *Infinite Mario* implements many elements of the original game, including Super Mushrooms, Fire Flowers, Goombas, Koopas, Spiked Koopas, Bullet Bills, and Piranha Plants. It is not a completely faithful replication, though, as it does not include some game elements, such as moving platforms or the Cloud Koopa character, who hurls Spiked Koopas from the sky. However, a large proportion of the game play is still intact, and, judging by its popularity on online gaming sites such as NewGrounds [33], where it has been played by over 11,000

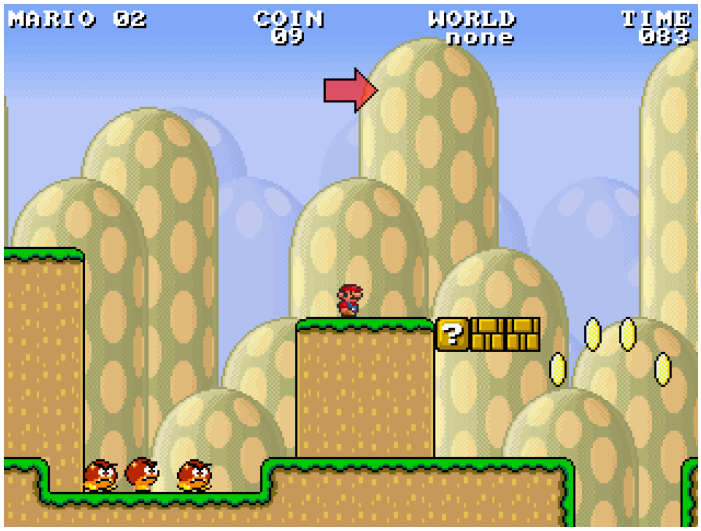


Figure 5.1: *Infinite Mario.*

people, it can be considered a legitimate representative of the platformer genre.

5.1 Challenge Metric

One should recall that the rhythm-group model serves as the core of the genetic algorithm, and, since the rhythm-group model is defined in terms of challenge, a method for estimating the challenge of a given level is required. In challenge-based games, difficulty arises from the precision required to execute a properly-timed sequence of button presses. In *Mario*, difficult segments correspond to the locations where enemies are densely located and where the platforms are narrow. Easy segments are, conversely, the areas where there is very little precision required to successfully traverse the section. More technically, if we consider the set of all possible sequences and timings of button presses, a challenging section can be defined as sections where the ratio between sequences of button presses that result in successful traversals to sequences that result in unsuccessful traversals is relatively small. It is this notion that motivates the metric of Compton and Mateas [10], in which the challenge of a particular jump is defined as the ratio between the number of trajectories that traverse the gap to the number of unsuccessful trajectories which result in falling into the gap. Our challenge metric is similar and is shown in Equation (5.1), where $d(p_1, p_2)$ is the Manhattan distance between the platforms p_1 and p_2 minus the sum of the two "landing footprints,"

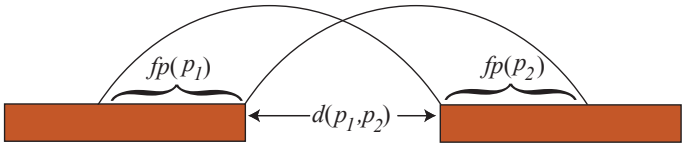


Figure 5.2: The landing footprint *fp* is a measure of a jump's margin of error.

fp, of both platforms plus a constant. The measurement is illustrated in Figure 5.2.

$$c(t) = d(p_1, p_2) - (fp(p_1) + fp(p_2)) + 2fp_{max} \tag{5.1}$$

The landing footprint is a measurement of the length of a platform, bounded to the maximum distance a player can jump, fp_{max}. This measure is important, as there is a much greater margin of error when jumping to a wide platform than to a narrow platform, and it is a less challenging maneuver. The constant $2fp_{max}$ is added simply to ensure that this difficulty measure is non-negative.

We extend this basic formula to account for the challenge posed by the enemies in the level. Because all enemies in *Mario* can be defeated by jumping on, or over them, we can regard each enemy as a gap in the level. We measure challenge in the same way as with platforms, save for an extra constant C_e, which is added to account for the fact that since the enemies are moving, there is a slight increase in difficulty as opposed to a static hole of the same size.

5.2 Design Elements

We conducted two separate experiments with different sets of DE definitions. The first set contains an assortment of basic DEs that describe only simple level components. The other set contains more complex DEs in addition to the simple ones, which represent more complex structures that can be recognized in many different levels from the original *Super Mario Bros.* game.

5.2.1 Basic Formulation

The basic formulation consists of the following DEs:

- *Block*(x, y). This DE is a single block, parameterized by its x and y coordinate.
- *Pipe*$(x, height, piranha)$. A pipe serves as both a platform and a possible container of a dangerous piranha plant.
- *Hole*$(x, width)$. This specifies a hole of a given width in the ground plane.
- *Enemy*(x). This specifies an enemy at the given horizontal location.
- *Platform*$(x, width, height)$. This specifies a raised platform of a given width and height.
- *Staircase*$(x, height, direction)$. Staircases are common enough to warrant a dedicated DE. The direction specifies whether the stairs are ascending or descending.

It is necessary to describe the translation process between the DE representation and the internal data format used by the game engine. An *Infinite Mario* level is internally represented by an array of 250×15 block units with a floor consisting of blocks that occupy the entire 14th and 15th rows of the array. Beginning and end points, which are required by *Infinite Mario* to determine the spawn position and victory criterion, are set to $(0, 13)$ and $(230, 13)$, respectively. The static elements that make up the level (namely, pipes, cannons, blocks, and ground) are described by this array. Each distinct type of static level element is represented by a unique ID number, and horizontal and vertical positioning within the level corresponds exactly to the columns and rows of the array. To translate a DE for one of these static level elements into the internal format of *Infinite Mario*, the system writes the corresponding ID number to the appropriate array cell. For example, given that the ID for a regular block is 16, we translate the DE *Block*$(5, 7)$ into a format compatible with *Infinite Mario* by writing the number 16 to the seventh row of the fifth column of the level array data structure. Notice that because levels contain a floor by default, holes must be created explicitly in the environment through DEs by writing the number 0, which denotes empty space, to the appropriate cells.

Enemies are represented as "sprite objects" and are stored in a linked list, separate from the static level structure. Each sprite object contains member variables that describe its type, as well as its horizontal and vertical position; thus, these objects are virtually identical to our DE representation. It is trivial to instantiate sprite objects that correspond to the non-static DEs and to insert these objects into the linked list.

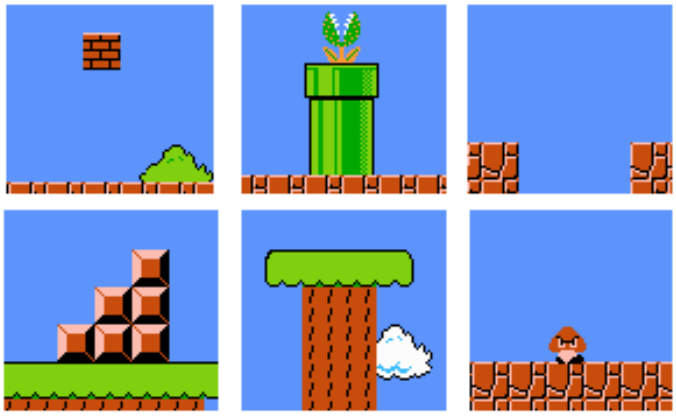


Figure 5.3: Basic *Mario* DEs. Clockwise from top-left: Block, Pipe, Hole, Enemy, Platform and Staircase.

We define, in addition to the DEs, a number of constraints that express the requirements for a playable *Infinite Mario* level. As previously noted, each constraint in our system can specify a method for penalizing levels proportionally to how greatly they violate the constraint.

- *require-exactly*$(n, type)$. This constraint specifies the desired number of certain types of design elements to be present in the levels. As a penalty, it returns the absolute difference between the counted number of instances of *type* and the desired amount n.

- *require-at-least*$(n, type)$. This function penalizes levels that contain less than n of a given *type*, returning 0 if $n \geq type$ and returning $type - n$ otherwise.

- *require-at-most*$(n, type)$. This function penalizes levels that contain more than n of a given *type*, returning 0 if $n \leq type$ and returning $n - type$ otherwise.

- *require-no-overlap*$(type_1, type_2, ...)$. This function states that the specified types are not to overlap in the phenotype. It is, therefore, only relevant for design elements that contain a notion of location and extent. In the present application, we specify that pipes, stairs, enemies, and holes should not overlap one another. As a penalty, the number of overlapping elements is returned.

- *require-overlap*$(type_1, type_2)$. Though similar to function 4, this function specifies that $type_1$ must overlap $type_2$, though $type_2$ need not necessarily overlap $type_1$. We use this

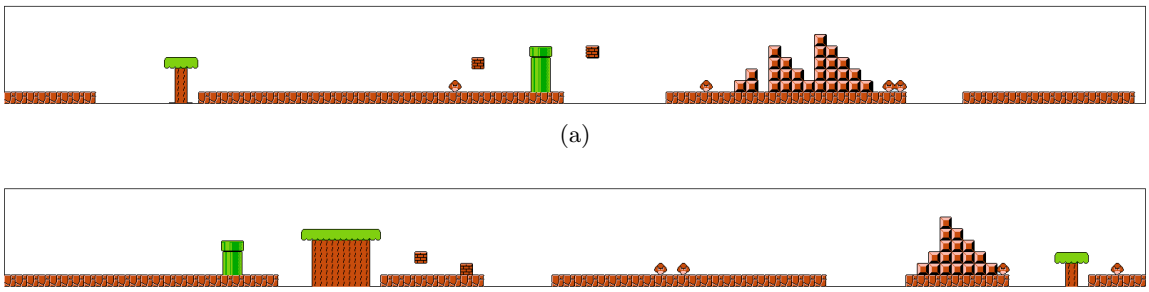
function to require that platforms must be positioned above holes. The number of $type_1$ elements that do not overlap with a $type_2$ element is returned as a penalty.

- *traversable*(). This function is to ensure that a player can successfully traverse the level, meaning that there are no jumps that are too high or too far for the player to reach. This is determined using a greedy search between level elements. The penalty is the number of elements from which there is no subsequent platform within a specified range, that is, the number of places at which a player could get stuck.

All the previous functions are specified such that a value of zero reflects a satisfied constraint, and a positive value denotes how severely a constraint is violated. These constraints can then be used as the fitness function for the infeasible population in the Tier 2 constraint satisfaction system. Therefore, any individual level that is given a score of zero by all of the above functions is considered a feasible solution and is moved into the feasible population for further optimization. The feasible population is evaluated using our generic model of challenge-based fun.

All these constraints, except for *traversable*(), are specified in terms of local, spatial geometric relationships between objects. This means that nearly all of these requirements can be satisfied immediately by the Tier 1 system, and, therefore, very few individuals need to be moved into the infeasible population. However, there are individuals that fail the traversability requirement or that contain more DEs than could possibly fit within the space limits of a single level. These individuals cannot be repaired by the Tier 1 system and must be placed in the infeasible population. As well, we find that it takes a great deal of time for the Tier 1 system to identify a level as non-repairable, as this, essentially, requires an exhaustive search. Because this slows down the entire run, we place a timeout on the Tier 1 solver, so that if an individual takes a significantly longer time to repair than the current average repair time, it is immediately moved to the infeasible population. Because repairable levels tend to be fixed relatively quickly (well below one second), the timeout serves as a fast way to identify potentially infeasible level designs. Even though some repairable levels could be misclassified as infeasible, they still can be repaired through the Tier 2 process, so this trade-off is reasonable.

With no pressing concern for efficiency, we choose to set the mutation rate to 10% of individuals per generation and to generate the rest via crossover, using tournament selection of size 3. Finally, following the convention of Kimbrough [24], we limit the sizes of the



(a)

(b)

Figure 5.4: Segments from final levels with the basic set of DEs.

infeasible and feasible populations to 20. The island model is not necessary in this case, as the population converges on acceptable designs without this extension. Our stopping criterion is attained if the fitness of the levels does not improve for 50 generations. Figure 5.6 depicts some resulting levels.

Evaluation

The generated levels are playable and violate none of the specified constraints. As well, the rhythm-group structure is clearly present, with periods of low difficulty separating enemy locations.

The designs from this formulation were informally evaluated during a technology demo held at the School of Interactive Arts and Technology's annual open house. The generated levels were played by several dozen attendees, consisting primarily of high school students. New levels were generated for each participant, though some levels were re-generated if they proved too difficult for the player to complete. Many played the levels to completion, suggesting a certain degree of engagement with the generated content. However, several individuals, especially those who professed considerable experience with *Mario*, commented on the lack of variety in the level designs.

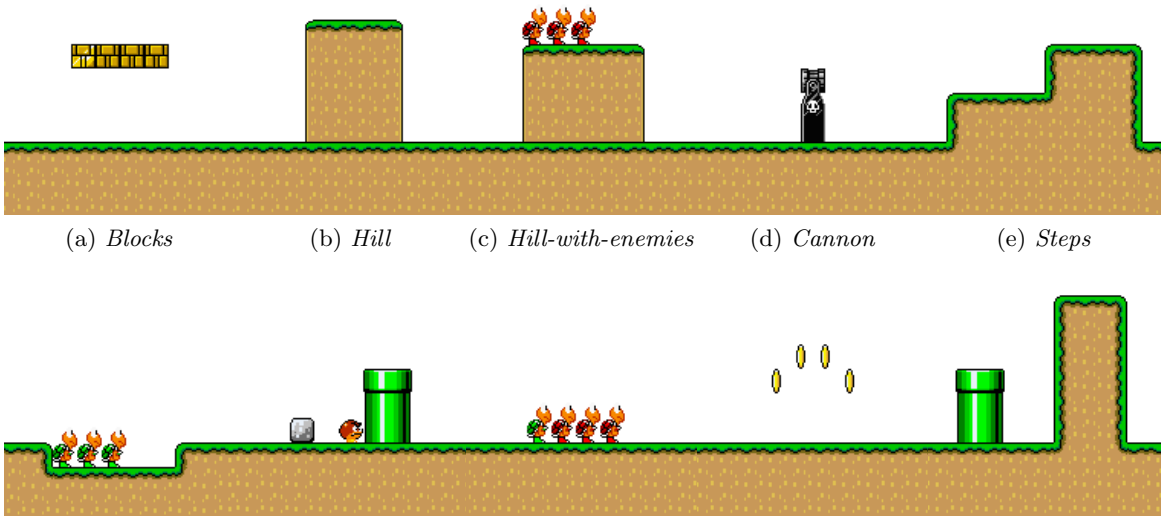
5.2.2 Complex Formulation

The complex formulation consists of ten additional DEs which are used in conjunction with the simple set. These elements are drawn from common patterns seen in both the original *Super Mario Bros.* as well as one of its sequels, *Super Mario World* [30], and many represent

compound arrangements of the basic DEs. The purpose of the extended set is to encourage a greater degree of variety in generated levels, and to produce levels that more closely resemble the content from the actual game.

- *Blocks*(x, *height*, *width*). Several blocks in a horizontal row.
- *Hill*(x, *width*). A "background" hill that can be either jumped upon or bypassed.
- *Hill-with-enemies*(x, *height*, n, *type*). A Hill with n enemies of the *type* variety.
- *Cannon*(x, *height*). A cannon fires "Bullet Bill" enemies toward the player.
- *Steps*(x, *width*, *height*, *dir*, n). This DE represents when the ground itself rises or declines (specified by *dir*) to *height* in n distinct steps.
- *Enemy-pit*(x, *width*, *type*, n). This specifies a group of enemies at the given horizontal location, situated in a depression in the ground.
- *Enemy-pit-above*(x, *width*, *type*, n, *left*, *right*). This specifies a group of enemies at the given horizontal location, bounded by rocks, pipes, or cannons, as specified by *left* and *right*.
- *Enemy-row*(x, *type*, n). This specifies a group of enemies at the given horizontal location.
- *Coin-arc*(x, *height*, n). This specifies a number of coins at the given horizontal location.
- *Impediment*(x, *type*). A high pipe or wall that can only be mounted by jumping from another pipe, a row of blocks, or a background hill. This DE constructs both the obstacle and the helper object needed to cross it.

No new constraints are necessary, as the six already specified are able to express the intended relationships between these new elements; we generally require that these DEs do not overlap one another, though some, such as the *Hill* and *Coin-arc*, are able to be placed on top of certain other DEs. Although there is certainly additional complexity involved in specifying more DEs, each element is completely independent and can therefore be developed and tested separately. As well, though the number of constraints between DEs could increase exponentially as more are specified, many of the new DEs are subject to



(a) *Blocks* (b) *Hill* (c) *Hill-with-enemies* (d) *Cannon* (e) *Steps*

(f) *Enemy-pit* (g) *Enemy-pit-above* (h) *Enemy-row* (i) *Coin-arc* (j) *Impediment*

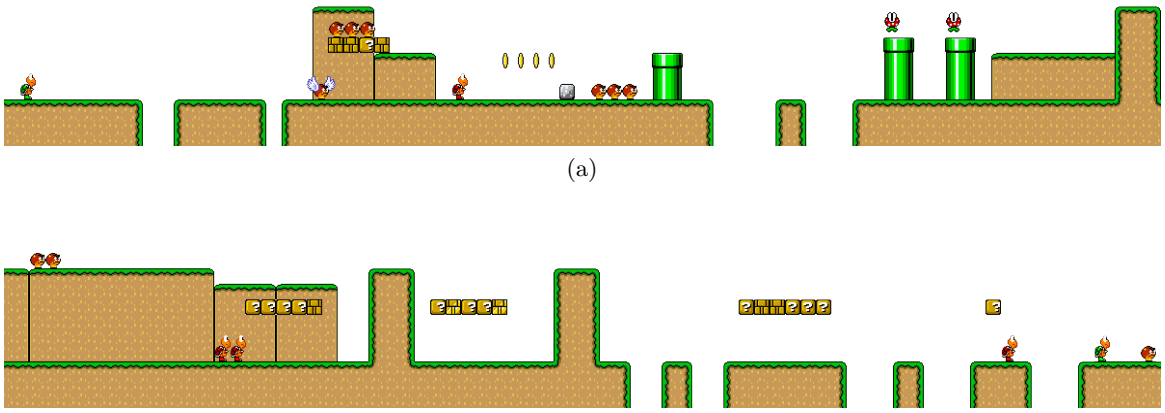
Figure 5.5: Examples of the various complex DEs.

identical constraints. For example, most of the DEs form a set of elements whose members cannot overlap with each other. Thus, it is not necessary to manually specify pairwise constraints between each DE.

The system is run with the same parameters as the simple set, with M equal to 6.0, unless otherwise stated. T_{window} is set to 9.4 and m is set to 0.5. Evolutionary runs take between two to ten minutes on a mid-range dual-core PC. Segments of some of the results are shown in Figure 5.6. It is evident that expanding the set of design elements results in levels that appear more intricate and more similar to levels produced by a human designer. Four levels generated by the system are shown in Figures 5.7–5.12, alongside representations of the modeled anxiety (accumulated challenge per rhythm group).

Figures 5.7–5.10 depict levels generated with M fixed at 6.0. These levels were the result of four contiguous runs of the system (*i.e.* they were not singled out according to subjective criteria). Graphs depicting the fitness improvement during the run are provided in Appendix B.

Figure 5.11 demonstrates the effect of varying M. By doing so, levels are created such that the most difficult portions are located where M is the highest, in this case, in the center of the level. However, some challenge is still present throughout the level, and the player is provided with constant engagement. This ability to alter the model's parameters offers



(a)

(b)

Figure 5.6: Segments from final levels with extended set of DEs.

designers a unique, high-level way to influence the system's output.

Figure 5.12 depicts another avenue for high-level control over the system output. In this case, a constraint was specified that there be no *Hole* DEs in the level genotypes. The system is able to produce rhythm-group structures while observing this externally imposed design requirement. By restricting the presence of certain level elements in this manner, we can ensure that levels do not all contain the same proportion of level elements and can therefore maintain diversity in the system's output.

As well, Figure 5.13 demonstrates the potential for our system to support a mixed-initiative approach to level design. A small segment of a level is designed by hand and converted to its DE representation. This translation is always possible for *Infinite Mario*, as every basic level component has a corresponding DE. This translated portion is fixed in every genotype of the evolutionary population and cannot be altered by any crossover or mutation operators; the hand-designed portion is kept separate from the rest of the algorithmically created DEs, and crossover and mutation may only occur on the latter set. Otherwise, with respect to evaluating fitness, the manually created portion is internally treated identically to the generated segments of the level and is subjected to the same constraints and fitness function. For this reason, the system is able to integrate human and artificial designs together in a manner that exhibits rhythm groups.

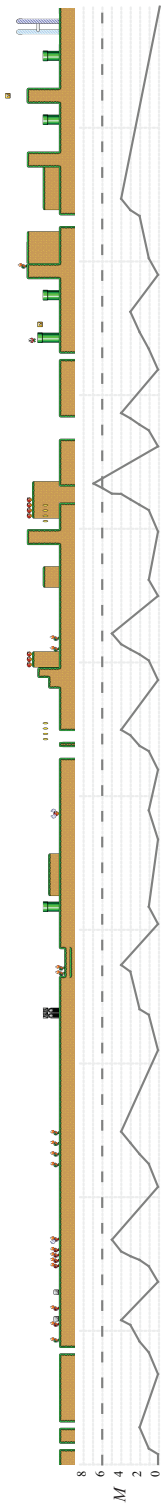


Figure 5.7: Fixed M. Generation 270, fitness 11.09

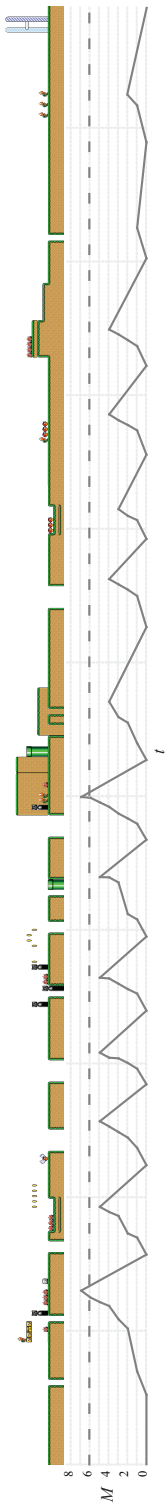


Figure 5.8: Fixed M. Generation 365, fitness 12.17

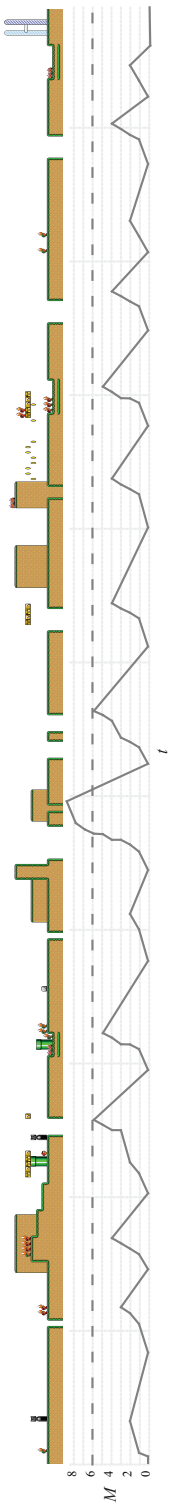


Figure 5.9: Fixed M. Generation 76, fitness 12.12

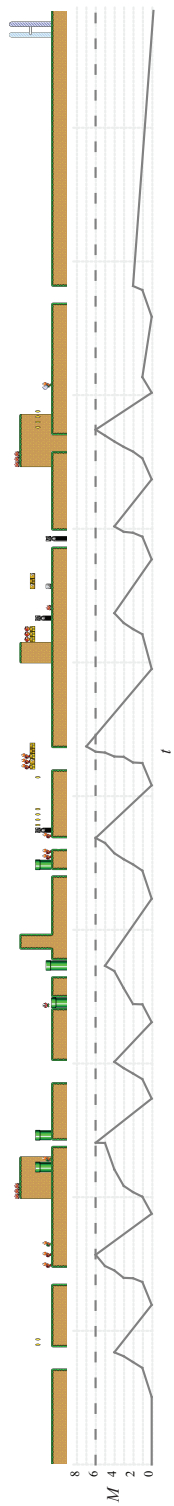


Figure 5.10: Fixed M. Generation 162, fitness 10.56

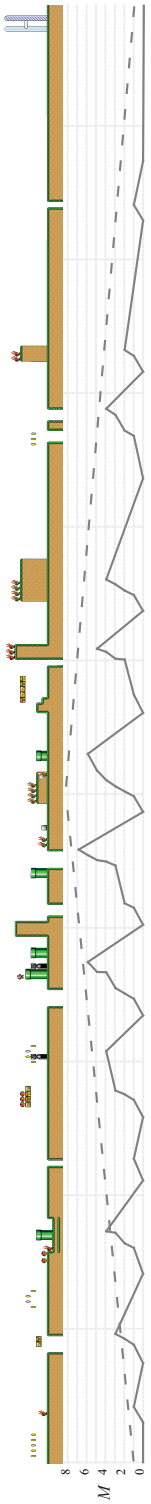
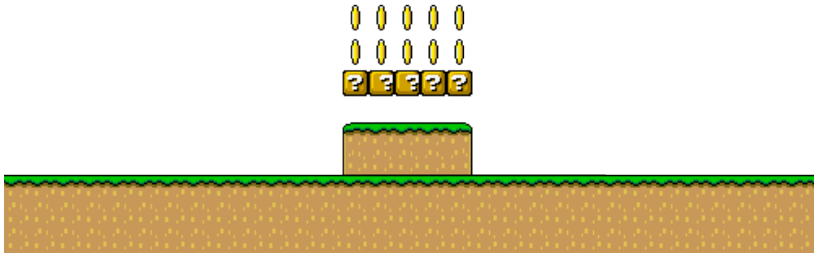
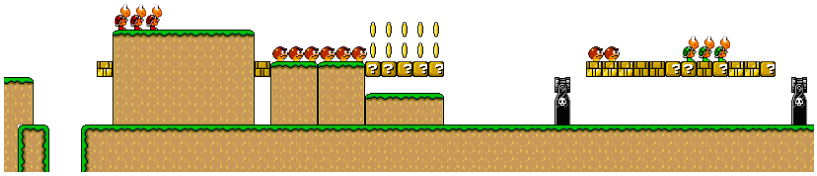


Figure 5.11: Varying M. Generation 629, fitness 10.55

Figure 5.12: Fixed M, with no $Hole$ DEs permitted. Generation 202, fitness 11.50



(a) Hand-specified design.



(b) Automatically-generated content added.

Figure 5.13: The evolutionary system adds content surrounding the human-specified portion. Notice the highly challenging portions on either end of the relatively simple middle section.

5.3 Mario AI Competition

This system was entered into the 2010 Mario AI Championship Level Generation Track, which was held at the 2010 IEEE Conference on Computational Intelligence and Games. Conference participants were invited to play levels generated by the various systems and to evaluate them according to how fun they were. The contest was arranged so that the systems had to produce levels that adhered to certain compositional requirements. For example, levels might be required to have four gaps and three Shelled Koopas. This requirement was put in place to discourage cheating through the use of systems that could merely return levels that were pre-designed by hand. To observe this rule, our system translated the composition requirements into constraints.

Fifteen participants took part in the event, and our system placed 3rd out of 6. This is an encouraging result, as our system, guided by a generic fitness function, was able to rank competitively with systems designed specifically for this particular game.

5.4 Discussion

One possible shortcoming of our approach is that the rhythm-group model cannot express diversity; there is no evolutionary advantage to producing rhythm groups that contain a mixture of different elements, as opposed to creating levels consisting of a single kind of element. For example, it is possible for a level containing only Koopa enemies to have the same rhythm-group configuration, and thus, the same fitness value, as a level containing a mixture of enemies and holes. This type of monotonous design does not seem to occur in practice because of the stochastic nature of the genetic algorithm. On the other hand, this restricted diversity in a level's design might be considered desirable. In this case, the constraint system could be used to influence the variety of the level designs by enforcing a maximum or minimum amount of certain game elements. For example, it is common to introduce particular enemies only in later levels of a game; the designer can set a constraint specifying that early levels contain no DEs of this type. Figure 5.12 depicts how this high-level control can be used to create a level with no holes. This control can ensure that the system produces levels that do not resemble each other but instead differ greatly in terms of composition and appearance.

Another potential criticism might be made against the complexity of some of the design elements. Indeed, reifying some structures, such as staircases, requires an imperative set of construction instructions to be specified. This type of bottom-up, procedural approach may seem out of place in a framework that purportedly minimizes such low-level, game-specific code. It is important to emphasize that while the method to build a staircase is, in itself, an imperative procedure, the instructions for doing so are parameterized: the overarching system can manipulate it on a high level by altering its height, width, and position, in the same manner as any other level element. In this sense, each DE is treated as a black box. This policy ensures that all the procedural knowledge contained within the DEs is insulated not only from the high-level system but also from other DEs; each basic level element can be developed and tested independently. Ultimately, we do not claim to completely eliminate all traces of imperative generative techniques; rather we recognize that a certain amount of procedural specification is necessary but restrict the scope of such techniques, subjecting them to manipulation by the high-level system. Complex components can be seamlessly used in conjunction with the rhythm-group fitness function, constraint solver, and mixed-initiative design approach in exactly the same way as any other DE. This flexibility is not typically afforded by monolithic, rule-based production systems.

Chapter 6

Evolving Zelda Levels

To support our claims of generality, we present an application of the generative system to an different game genre. In this section, we target game levels that consist of rooms and doors arranged in a two dimensional space and are viewed from an overhead perspective, as opposed to a side perspective as was the case in the *Infinite Mario* levels. The purpose of this simplification is to focus on the core problem of designing two-dimensional levels, as opposed to one-dimensional designs. It proves significantly more difficult to generate levels for this domain, but the fact that our approach is still able to efficiently create feasible solutions that exhibit a rhythm-group structure illustrates both our system's generality and its promise as a practically usable technique.

6.1 Game Background

Zelda is a series of action-adventure games developed and produced by Nintendo, which centers around the adventures of Link in the kingdom of Hyrule. In the course of a typical *Zelda* game, Link must successfully overcome the challenges of several dungeons. Each dungeon adheres to a recognizable pattern and consists of an arrangement of rooms filled with enemies, collectible items, and puzzles. Because finding an optimized arrangement of rooms is considered a difficult challenge for heuristic searches, our initial attempts to model this game involve significant simplifications. However, promising results in the simplified domain justify further experimentation.

Figure 6.1: Segment of the "Crystal Dungeon" from *The Legend of Zelda: A Link to the Past*.

6.2 Design elements

The following DEs are sufficient to examine the problem of two-dimensional room layout as it relates to challenge dynamics:

- $Room(x, y, w, h)$. A room of dimension $w \times h$ with its origin at the point (x, y).
- $Door(x, y)$. A door located at the point (x, y).
- $Enemy(x, y)$. An enemy located at the point (x, y).

6.3 Challenge metric

The challenge function is defined in terms of the enemies the player faces as they move from room to room. Rooms can be viewed as sets of the entities they contain, so given a player who enters $Room(t)$ at the moment t, the challenge at that point is defined as the number of enemies in that room, or, more formally, $c(t) = |\{e \in Room(t)\}|$. To simplify calculations, we assume the player enters a new room at each integral time step. It would be straightforward, however, to derive a more accurate time estimate using the euclidean distance between doors as a way of better approximating the time of entry into each room.

This formulation inherently presupposes that the player's path through the rooms is fixed and that $Room(t)$ represents a single value for each point t. We pick the shortest-path movement between the entrance point and the exit point as the canonical path of the player. The entrance and exit points are explicitly defined before evolution. Certainly, this is a

large simplification of the behaviour a human would actually exhibit when moving through a complex virtual environment. However, our choice of shortest-path can be justified by ensuring that levels contain only a single, unique sequence of connected rooms from start to finish. In levels such as these, any path which does not double-back on itself will be the shortest path. One can ensure that the generated levels contain no multiple paths or dead-ends, and thus adhere to this linear topology, by removing all but one connected path as a post-processing step.

6.4 Constraints

The Tier 1 constraints that can be solved in terms of a constraint satisfaction formulation, are as follows:

- *Room location and dimension values must be multiples of* 16. This constraint is equivalent to snapping the rooms to a coarse grid with cell sizes of 16 units, which simplifies detecting the property of room adjacency.
- *Doors must exist on room edges.* This constraint ensures doors do not exist, for example, in the center of a room. Enforcing this constraint involves snapping each door to its nearest wall segment.
- *No overlap between rooms.* Enforcing that rooms cannot intersect one other is solved through the manipulation of the shape and position of the rooms. This is an example of a two dimensional geometric packing problem, which is more difficult to solve than the one dimensional constraints used in the *Mario* application. However, the constraint solving library JaCoP provides geometric extensions that are able to efficiently solve this type of problem.

There is only a single Tier 2 constraint that cannot be expressed as a simple CS problem:

- *Connected path from the start point to the end point.* Two rooms are considered connected if they share a common edge and there is a door located on that shared edge.

To guide infeasible levels toward connectivity, we use the heuristics listed below. The heuristics are listed in order of the priority they are given when sorting the level designs.

In other words, any level which receives a higher value under Heuristic 1 will be given a higher fitness than any individual, irrespective of their Heuristic 2 valuation. Similarly, any individual which maximizes Heuristic 2 will be favoured without regard to Heuristic 3. Note that, in this application, it is more convenient to specify the heuristics such that positive values are desirable, as opposed to the FI-2pop convention of positive values representing penalties.

- *Room on start or end points?* This heuristic detects if a room exists at the beginning and ending locations of the level, returning 0 if neither is the case, 1 if one point is covered, and 2 if there is a room on both points.
- *Max path length from start and end.* This heuristic sums the maximum distance that can be traveled from the start point and the maximum distance that can be travelled from the end point, without backtracking.
- *Max path length from other rooms.* This heuristic sums together the maximum distance that can be traveled starting from every room in the level, other than the start and end point.

These heuristics serve to guide the infeasible population toward feasibility. By encouraging the maximum possible travel distance, the GA favours designs that attempt to connect multiple rooms together, increasing the probability that a design will be found that connects the beginning and end points together. This extra guidance is necessary, due primarily to the significant increase in the size of search space that the extra spatial dimension implies; every DE has two significant dimensions, x and y, instead of just x. Furthermore, a typical *Mario* genotype had approximately 40 DEs, whereas the *Zelda* levels we considered contained between 60 and 100 DEs. These two factors lead to a drastic increase of dimensionality, resulting in levels that would not achieve connectivity without some form of heuristic guidance. In a sense, these heuristics serve as an approximation of fitness on partial levels; instead of assigning a value of 0 to every unplayable level, we determine how much of the level is traversable (by measuring how far one is able to walk from a given starting point). Therefore, these heuristics are not entirely ad-hoc and unmotivated, but serve as a kind of adaptation of the fitness function to the case of broken levels.

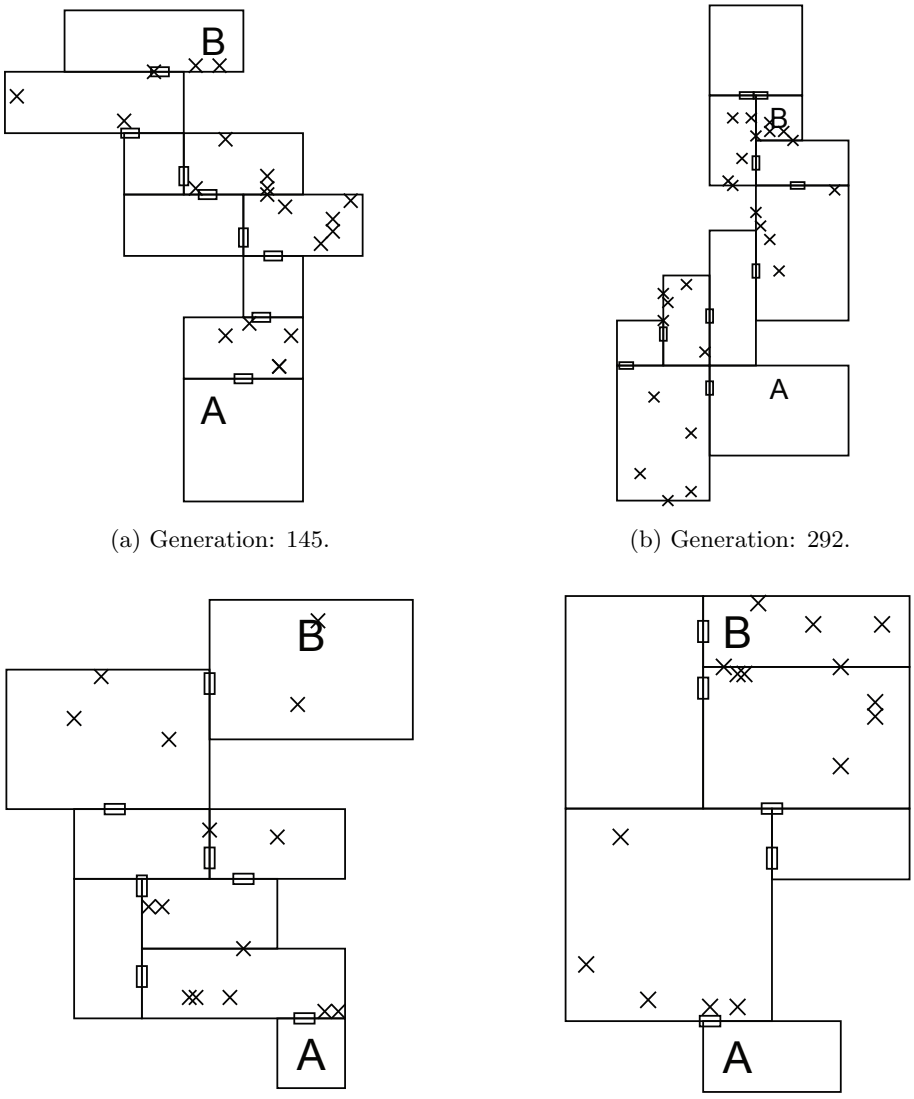
It should be mentioned that there is no explicit constraint that enforces the existence of a unique linear path through the level. Instead, this property is achieved through a

post-processing step. We can create a minimal genotype by attempting to remove each gene in turn, and replacing that gene if its removal affects the level's fitness value. Because we only represent the player's idealized movement in the $c(t)$ calculation as a single path, branches and dead-ends cannot possibly contribute anything to the fitness of a level, and are therefore always removed by this operation. Doors that lead nowhere are also removed from the genotype for this reason.

6.5 Results

Because it is difficult to arrive at feasible levels in this context, care must be taken when selecting the parameters for the evolutionary algorithm. For generating *Mario* levels, standard GA parameters were sufficient to enable reasonable evolutionary progress. However, these same parameter settings prove ineffective in the present case. To find useful values, several evolutionary runs were conducted under a number of different parameter settings, and the parameters which consistently produced the highest average rate of fitness improvement were selected. Evolution was conducted with 5 islands, each containing 20 feasible individuals and 20 infeasible individuals. Crossover was found to be effective at a rate of 0.7, with mutation at a probability of 0.03 per gene. Tournament selection of size 3 was used. There was elitism of a single individual, and inter-island migration was conducted every 50 generations. The stopping criteria is a lack of progress in all of the populations for 20 generations, as it was rare to witness any improvement after this point.

Results are depicted in Figures 6.2, 6.3 and 6.4. The rhythm pattern structure is evident in all results, with highly challenging, enemy-filled rooms separated by empty rooms. It is also obvious that all the levels satisfy the connectivity constraint; as expected, there is a single, unbroken path between the start and end points. The evolutionary runs took between 15 and 45 minutes on a mid-range dual-core PC, except for the level depicted in Figure 6.3, which was halted after 5 hours. Graphs of the fitness improvement are provided in Appendix B. It is somewhat expected that increasing the area of the space increases the number of generations required to find a good layout. However, this increase in generations is not proportional to the increase in time required, signalling that the amount of effort to process each generation is significantly higher. The reason for this likely lies in the constraint satisfaction subsystem, which must expend a great deal more time ensuring the rooms do not overlap. As well, it must be noted that the level designs are depicted in their reduced form, with all branches and non-essential genes removed; there were likely many more DEs



(a) Generation: 145.

(b) Generation: 292.

(c) Generation: 169.

(d) Generation: 284.

Figure 6.2: *Zelda* results, within an area of 224×224 units.

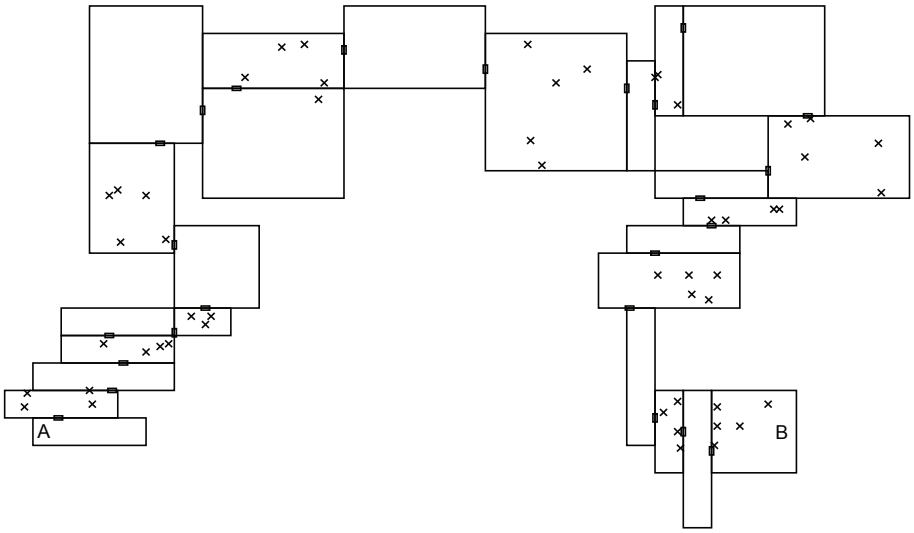
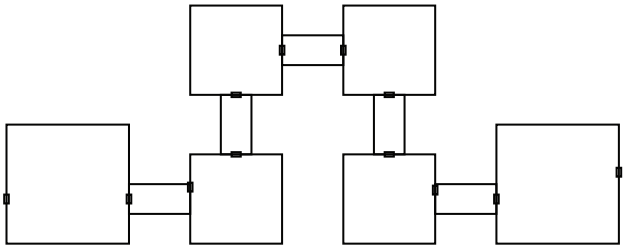
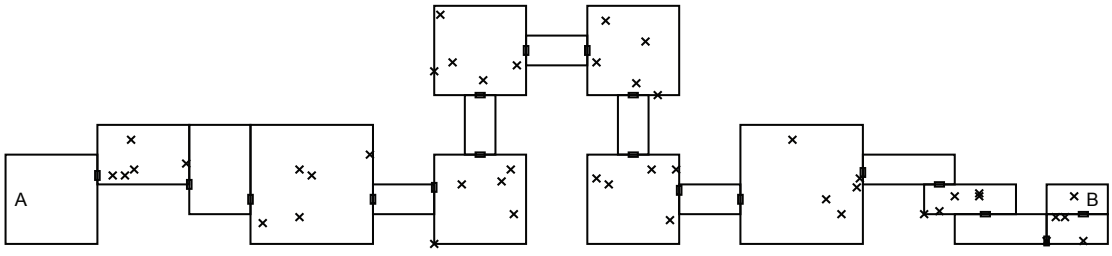


Figure 6.3: Generation: 731. Fitness 11.52. Evolved within an area of 576×576 units.



(a) Fixed genes; specified by human designer. No enemies are present.



(b) Enemies placed by system. Rooms also connected to entry and exit points. Generation: 517.

Figure 6.4: Mixed initiative design in the context of *Zelda*. Evolved within an area of 688×368 units.

processed internally than were present in the final layout.

Finally, Figure 6.4 depicts the result of mixed-initiative design. This design evolved relatively quickly (within 45 minutes) despite the large size of the area, likely because many of the rooms were specified by hand and the system primarily needed to adjust the location of enemies. Clearly, more work is required to identify where the computational bottleneck is located so larger levels can be produced. However, our specification in terms of rooms and doors as independent DEs is quite naïve, and there is a large number of improvements that could be made to the representation. The fact that recognizable rhythm structures can be seen in this sub-optimal problem specification provides evidence that further research is worthwhile.

6.6 Discussion

6.6.1 Future work

The *Mario* context began with a minimal set of DEs, and was expanded incrementally with more interesting, complex DEs. This process would certainly aid in reproducing the look of actual *Zelda* levels as well. As well, constraints that model the puzzle aspects of the game could be considered. For example, keys must be attained to access various parts of the dungeons that provide the player with various items needed to successfully defeat the dungeon's final boss. These complex requirements seem well suited to being specified as Tier 2 constraints, in a manner similar to the connectivity constraint. Instead of checking the single path from the beginning to the exit, one could check the path from the entrance to the first key, then to the second key, and so on. It would also be interesting to consider levels containing multiple paths, instead of requiring singly-connected paths. It might be possible to allow for multiple paths by aggregating together the results of running each possible path independently, via the mean, minimum, or maximum output from the model.

6.6.2 Comparison to *Mario* results

Though the generated levels appear simpler in appearance than the generated *Mario* levels and the formulation contains fewer distinct DEs and employs a simpler challenge function, it is, in reality, far more difficult to generate a feasible level for this context than for *Mario*. The combined factors of the higher dimensionality and the connectivity constraint render it virtually impossible to produce a feasible level by chance alone. Indeed, there is a vastly

greater number of possible broken levels than feasible levels; Compared to the *Mario* application, where 195 out of 1000 randomly generated levels prove to be feasible with no need for further constraint satisfaction, the *Zelda* application produces no feasible individuals at all from chance alone.

The minimal representation adopted in this context was likely responsible for the high computational cost of evolving levels. Level connectivity had to be attained through the use of two different types elements (doors and rooms, corresponding to edges and nodes in graph terminology) aligned in a fragile configuration. These difficulties were not present at all in the case of *Mario*, where an empty level (consisting only of the floor, from beginning to end) was still considered feasible. A *Zelda* representation which treated doors as a dimension of a room DE (for instance, as Boolean flags determining if there are doors on certain sides of a room) would reduce the number of different types of elements that would need to be aligned to achieve connectivity, and therefore reduce the search space significantly.

In any case, it is clear that compound DEs and complex challenge metrics are not necessarily responsible for the difficulty of generating levels for a given game; it appears that the nature of the constraints influence the difficulty of generation to a far greater degree. For this reason, it is probable that more DEs and a more precise challenge metric could be developed to allow the system to generate more interesting Zelda levels, so long as the connectivity constraints do not become more difficult to satisfy. There is a likely a reasonable flexibility within the design space provided by the existing design constraint of single-path connectivity. It is even possible that single-path connectedness is a more fragile, and hence a more difficult, constraint to satisfy than constraints over multiply connected spaces, and levels that admit multiple paths might prove to be easier to generate.

The results of this intentionally simple design context provide evidence that this generative system can be successfully applied to contexts quite different than 2D platformers. This application is not intended to demonstrate the maximal amount of detail and complexity that can be produced by the system (which was demonstrated more effectively in the context of *Mario*) but rather to determine the system's ability to satisfy difficult design constraints in a different game genre. Indeed, by using exactly the same fitness function and genetic operators (with some parameter adjustments) viable levels were generated in a significantly different context while maintaining all the advantages of the top-down approach: the model offers high-level parameters which serve to alter the layout of the rooms to provide higher or lower levels of difficulty, and the system is still compatible with mixed-initiative design in that the environment can adapt itself to fixed components specified by a human designer.

Chapter 7

Conclusion

By applying our approach to different types of games, we have illustrated the generality of our system. This generality exists along two dimensions: the rhythm-group can be applied to different game contexts, and the generative system is likewise able to create levels for games from different genres. We do not claim that this generality comes "for free"; certainly, some effort is required to apply the system to a new context. To leverage the rhythm-group model, a function $c(t)$ must be specified in order to provide an estimate of the challenge dynamics for a level. We argue, however, that the specification of a challenge estimate is easier than the specification of level's entertainment value. Like Hunicke et al. [21], we believe the notion of 'fun' is too broad to yield concrete design guidance, and more precise terms are required to motivate the design process. Thus, through the use of our model, the act of determining level quality does requires that the designer supply not a measure for the vague concept of 'fun' but rather an estimate for the much more concrete notion of 'challenge.'

Similarly, a sufficient set of DEs and constraints must be defined before the system can generate new levels. We propose that this requirement, too, presents a reasonable cost. DEs represent the basic components of a game's environment, and many of these components are likely to have similar representations within the game's existing design tools. For example, a human designer would need to be able to place blocks, platforms, and enemies when creating *Mario* levels by hand; therefore, these objects would need some sort of explicit representation. Indeed, there is a natural correspondence between our representation, which consists of DEs with varying dimensions, and the object-oriented paradigm, in which objects possess internal member variables. This object-oriented architecture is typical of

current game engines (for examples, see the *Unreal Development Kit* [14] or the *Unity Game Engine* [58]). The specification of design constraints, however, is not generally practised in current game development, though it is possible that doing so would provide secondary benefits beyond allowing for the procedural generation of new levels. By specifying explicit playability constraints, the designer might be able to reduce the number of bugs in games environments, and the need to expend effort in play-testing levels would be lessened. However, even if this benefit alone is not enough to justify the effort required to define explicit playability constraints, some form of constraint specification is necessary before procedural generation can take place. This specification can be explicit, as is the case with our approach, or implicit, through the careful construction of production rules or grammars that produce only playable designs. Our approach constitutes a way of declaratively, rather than imperatively, specifying generative knowledge (i.e., how much challenge is enjoyable, which level arrangements are traversable, and so on). Instead of considering design goals to be an incidental side-effect of the generative procedure, the constraints and fitness function provide an avenue for explicitly representing these goals.

We envision that our approach could usefully function as middleware in the game development process. Developers could incorporate our tool into their workflow in much the same way as is currently done with third-party 3D engines or physics libraries. Middleware is not necessarily intended to be used without any modification, but is instead meant to provide a reusable functional core that can be flexibly adapted to a wide variety of game contexts and extended as required. In this manner, the core functionality of our approach, which blends genetic algorithms and constraint satisfaction with a generic model of challenge-based fun, could be adapted by developers to suit their particular needs. The developers would supply the set of DEs, the constraints, and a heuristic for measuring challenge, and they could make use of the generative system as an external library. Furthermore, it is quite possible that many constraints are similar between games, especially games within a similar genre. For example, some form of connectivity constraint would likely be shared by many games involving the traversal of space. These common constraints could be included with the generative system, and developers would need only to specify the constraints that are not provided with the system and are unique to their game. Using our system in this manner would further reduce the amount of work needed to implement an automated content generation system for a particular game context.

Our approach confers further benefits in addition to the low cost of specification. Because it is a fundamentally top-down architecture, designers can influence the production of levels at different levels of abstraction. At the highest level, designers can vary the model parameters to produce levels with different challenge profiles. By adjusting M, it is possible to create levels that exhibit a dramatic arc, with a low-intensity beginning, an extremely difficult climax, and a relaxing denouement. As well, designers can produce a series of progressively difficult levels which presents players with continuous challenge as their skills increase. At another level of abstraction, our system's explicit constraints allow designers to exert control over the composition of game levels. It is possible to specify that the first few levels contain only a few different types of enemies and that more diverse combinations of elements occur only later in the game. By allowing control over the exact combination of DEs that compose particular levels, designers can fully exploit the full range of interactions between the game's various elements. At an even more fine-grained level of abstraction, designers can specify certain portions of levels by hand, and the system can seamlessly adapt content to these segments. This allows for the existence of level constructions that would not occur naturally under the stochastic process of evolution, such as complex geometric arrangements of level components or periods of unusually high or low challenge. Such interventions on the part of human designers could help the system avoid producing environments that are indistinguishable and lacking in character.

7.1 Future Work

The work outlined in this thesis is preliminary and there are certainly many promising avenues for further research. An obvious extension would be to apply our generative technique to even more types of games. Arcade games that follow the player character moves through space, such as "shoot-em-ups," in which the player pilots a ship that moves through various waves of enemies, would likely be quite amenable to the challenge modeling techniques described in this thesis. Another relevant genre to which our system could be applied is first-person shooters. The level layouts for such games could be generated in a manner similar to the dungeon designs seen in the *Zelda* application. This could prove especially lucrative for large, open-world games where a large amount of content is required.

Arcade games such as *Breakout* [6] and *Space Invaders* [34] are also potential candidates. The levels, which consist of arrangements of blocks and enemies, respectively, could be

readily described in terms of DEs. However, a suitable challenge metric for such arcade games would need careful consideration, as these game spaces are not traversed in the same way as the platformers or adventure game genres that we have discussed. Instead of moving through a relatively static environment of varying difficulty, the player character (in the form of the *Breakout* paddle or the *Space Invaders* space ship) remains relatively stationary in a dynamically changing environment. The estimation of challenge would likely need to adopt a more statistical approach, perhaps through a simulation of the progressive dynamics of the environment over time.

Indeed, a more thorough consideration of the topic of challenge estimation in general would be valuable to our approach. It would be beneficial to our goal of broad applicability if we were able to computationally model challenge in the same generic way as we currently model fun. There is research being conducted on the automatic detection of player frustration in games [22], as well as on statistical methods for modeling player preferences [60, 37, 44], and these efforts could prove complementary to our current approach.

An important aspect of challenge-based games is collectibles, such as coins, power-ups, and health packs. These provide advantages to the player and clearly influence the challenge dynamics, albeit in an indirect manner. A level with a large number of health packs would certainly be easier than a level that contains none. It may be possible to model these in the current system as presenting a negative amount of challenge, offsetting the positive challenge introduced by enemies and other obstacles. We have not yet explored this possibility.

It would also be desirable to more thoroughly test the model. Though we have validated the model against real-world *Mario* levels, and indirectly evaluated system's output at the Mario AI Challenge, it would be interesting to see if there was any discernible correlation between what actual players found fun and what the model predicted as fun. Participants could play a collection of real-world and automatically-generated levels and report how much fun they experienced. Ideally, we would hope to observe a statistically significant positive correlation between the subjective evaluations of the various levels and our model's predictions. Similarly, one could attempt to train the model not on positive examples taken from commercial games, as was the case in this work, but instead according to a player's individual preference, as expressed through his or her subjective evaluation of play.

As well, because of its top-down design, our generative system does not necessarily

exclude other procedural generation techniques. While we have not yet explored this possibility, our system could possibly be used in conjunction with other content creation techniques. Certain segments of a level could be generated by other algorithmic approaches and treated in the same manner as our system currently adopts toward hand-designed content, adapting around it in a manner that observes global design constraints. Another possible approach would be to employ our system as a method for controlling various sub-systems. If a particular generative grammar, for example, could be parameterized through a number of adjustable variables, it could be treated as a DE within our framework; the overarching genetic search would then attempt to find parameters for the generative grammar that produce levels with good rhythm-group patterns. This approach would be most applicable if certain procedural techniques could be tasked with the construction of specific level patterns; in much the same way as DEs can represent compound level components (such as "stairs" in our *Mario* application), bottom-up, rule-based subsystems could produce compound level elements but still remain under the control of the top-down, genetic search. For this reason, our approach does not need to be a monolithic framework but could be seen as a flexible, integrative approach.

7.2 Summary

We have presented a comprehensive approach to the generation of videogame levels which is founded on an explicit model of the relationship between challenge and fun. The model, based on the notion of rhythm groups, identifies fun as a function of challenge according to an "inverted-U" shape, as inspired by the Yerkes-Dodson law. The model's effectiveness was evaluated by employing it in a classification task, in which it was able to identify levels from the original *Super Mario Bros.* with a 97 percent accuracy.

Our model provides parameters that correspond directly to intuitive concepts. In particular, the parameter M corresponds to the skill of the player and can be adjusted by the designer to create levels with various challenge profiles. This high-level control is not typically offered by ad-hoc, bottom-up approaches where the relationship between the generative parameters and the final output is not always clear. Furthermore, human designers can directly specify certain portions of the level by hand, which are then evaluated by the model in the same manner as the automatically-generated content. This results in the rhythm-group structure adapting itself to the manually created portion of the level; difficult

portions are surrounded by easy sections, whereas simple stretches are surrounded by areas of high challenge. This natural adaptation to externally provided content is afforded by the top-down design of the system.

We described how our model is used as a fitness function in a genetic algorithm that guides a population of potential level designs toward levels which exhibit desirable rhythm-group structures. Our notion of DEs provides a flexible way to construct a genetic representation of a game level. We introduced corresponding mutation operators and crossover operators that preserve gene linkage in one or two dimensions. We also outlined a two-tiered approach to constraint satisfaction, with the first tier repairing geometric constraint violations between pairs of objects and the second tier handling levels that could either not be quickly repaired by the first tier, or which violate constraints that are inexpressible in terms of simple geometric relationships.

We have also demonstrated the generality of our approach in applying the model to two different games. It was possible in both cases to express the design problem in terms of a set basic DEs and a collection of geometric constraints. We also noted that the size and complexity of the DE set does not necessarily mean that levels will be more difficult to generate. Although the *Zelda* formulation had a much simpler representation than the *Mario* formulation, it proved more challenging to produce feasible levels for *Zelda*. More importantly, the rhythm-group model was applicable in both cases and produced visible recognizable oscillations in level difficulty.

In a broad sense, we consider this work to represent not only an effective way to generate usable procedural content for videogames but also a productive method of exploring theoretical accounts of game design. Our goal is to demonstrate that the exercise of specifying explicit models and evaluating them against real games reveals new insight into fundamental game design principles. It is our hope that modeling challenge dynamics in a high-level, generic manner will not only improve the quality of procedurally-generated content but also contribute toward further research in the analysis and understanding of the nature of fun in videogames.

Appendix A

Model Classification Results

Group	T_{window}	M	m	Misclassified
1	11.012454662897206	8.139846568797555	1.4082522929509942	1
2	10.887084140412833	7.985878659953309	1.2697402724677114	0
3	10.858950262083136	9.079084883492177	1.6825715761466602	0
4	10.756735695299048	7.998644987204351	1.871145924023131	0
5	10.715040398757896	7.909860471069911	1.2673145266369432	0
6	10.0306549548755	8.03018779366366	1.7078906764207409	0
7	10.90830238840967	7.926806884491277	1.653262753444488	0
8	3.3195099443994485	4.083626021703731	0.3858743228181518	0
9	10.411626263274878	8.1903409556286	1.9770030503485507	1
10	6.436891636930612	3.57088123312481	1.9155305995613865	0

Table A.1: Optimal model parameter settings for each of the 10 subsets of the cross validation experiment, as described in Section 3.5. The parameters were trained on subsets the 58 test cases (28 positive examples, taken from *Super Mario Bros.* and 30 hand-designed "poor" level designs). Each group was trained on 52 or 53 test cases (stratified, to include roughly equal examples of positive and negative in each training group), and evaluated against the remaining 5 or 6 test cases that were not used to train that group. There is a total of 58 evaluations and a classification success rate of 0.9655.

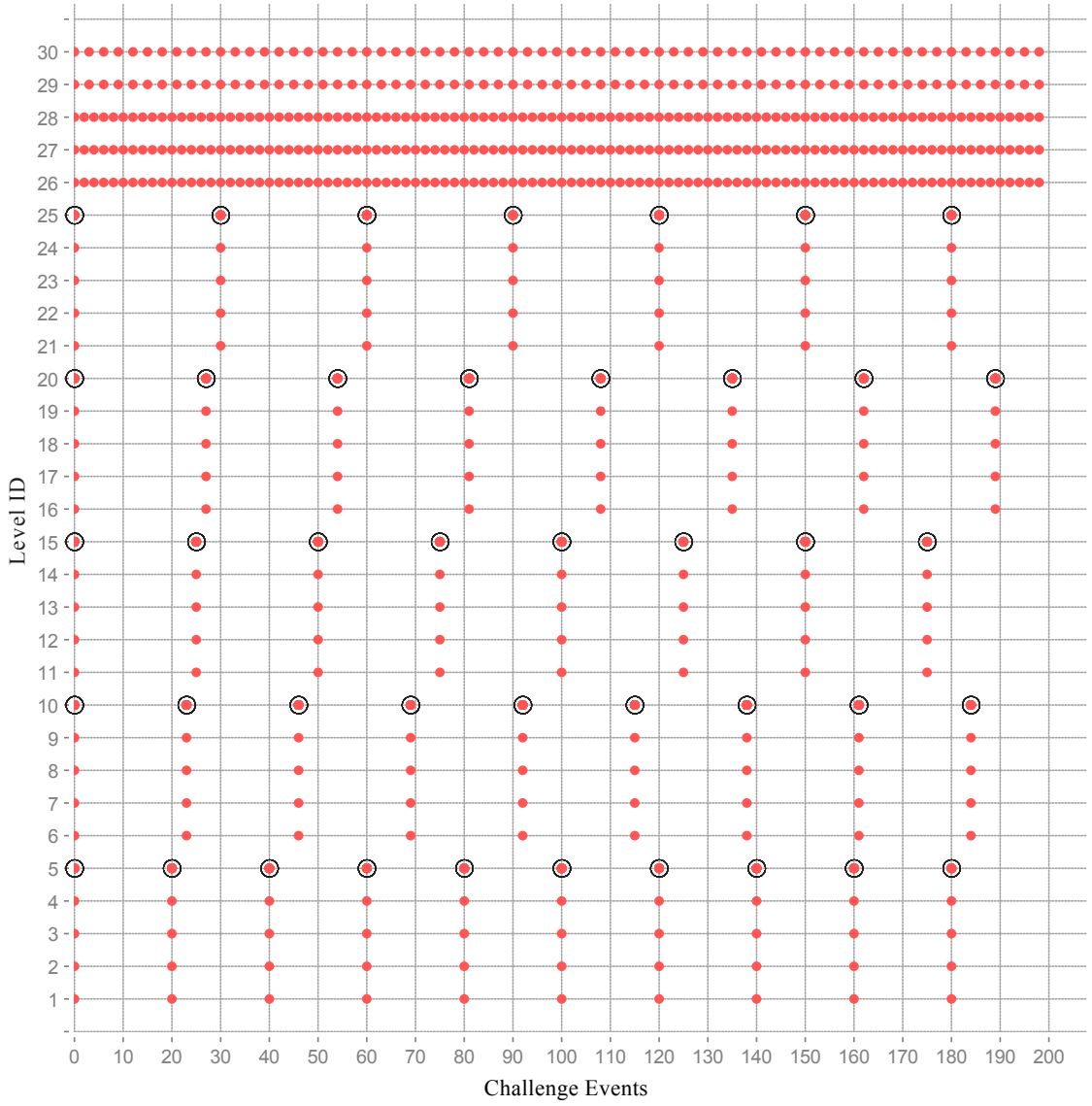
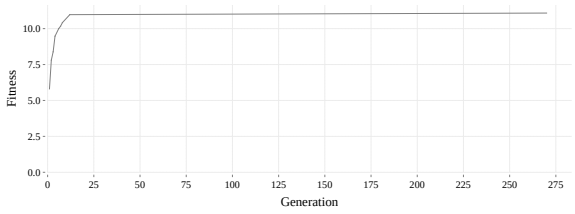


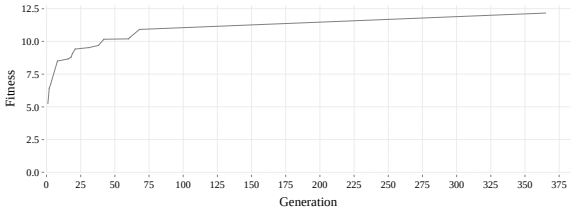
Figure A.1: Depiction of the challenge events of the 30 levels used to represent poorly designed levels in the model classification experiment. Horizontal scale is in 16×16 block units. Points of challenge that are encircled (as seen in levels 5, 10, 15, 20, and 25) indicate that the point consists of 20 challenge events located in the same position. These five levels, as well as levels 26–30, are constructed to represent levels that are clearly too difficult. The rest of the levels are indented to represent levels that are far too easy.

Appendix B

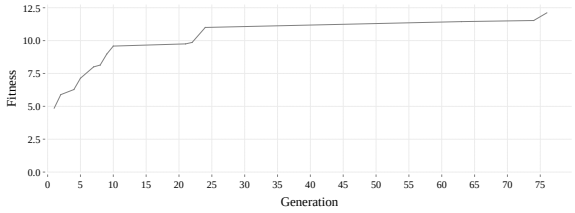
Evolutionary Run Results



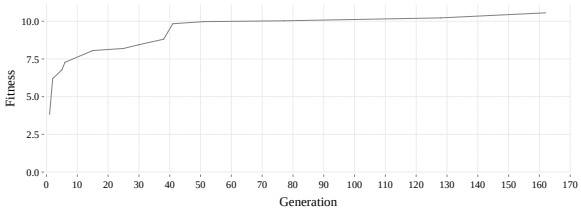
(a) Fixed M. Generation 270, fitness 11.09



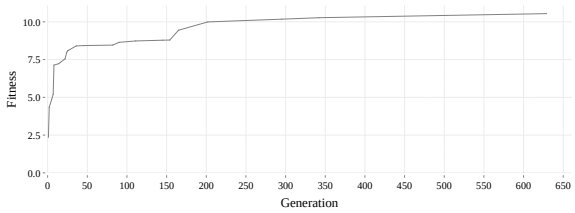
(b) Fixed M. Generation 365, fitness 12.17



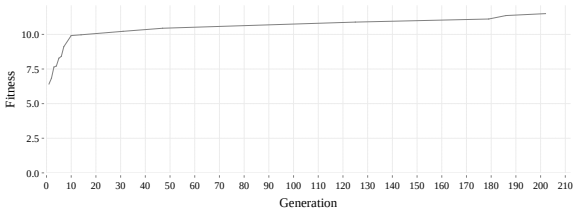
(c) Fixed M. Generation 76, fitness 12.12



(d) Fixed M. Generation 162, fitness 10.56

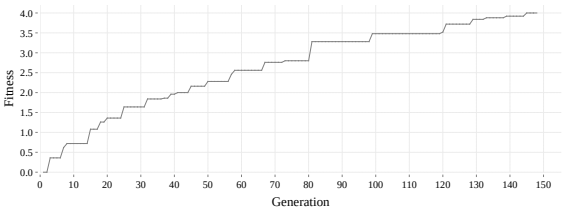


(e) Varying M. Generation 629, fitness 10.55

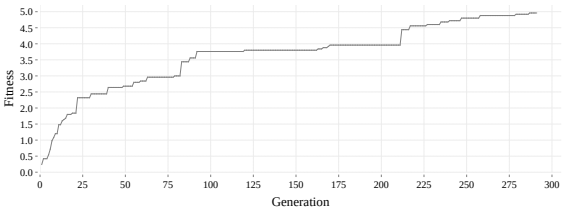


(f) Fixed M, with no *Hole* DEs permitted. Generation 202, fitness 11.50

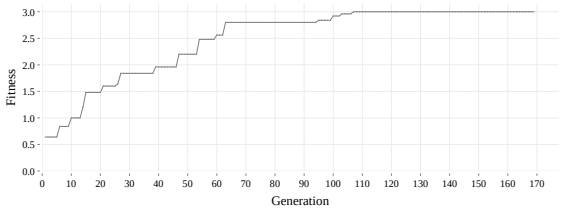
Figure B.1: Fitness improvement for *Mario* levels as discussed in Section 5.2.2



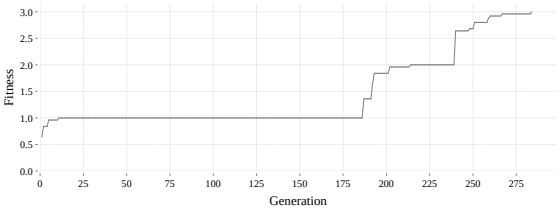
(a) Generation: 145.



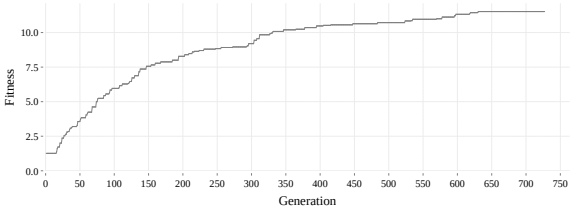
(b) Generation: 292.



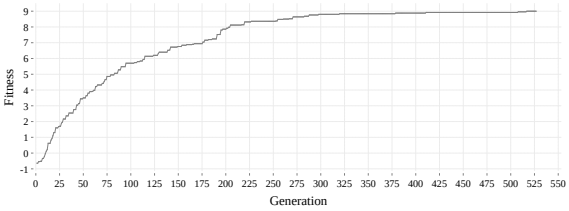
(c) Generation: 169.



(d) Generation: 284.



(e) Large level, size 576×576. Generation: 731.



(f) Mixed Initiative Results. Generation: 517.

Figure B.2: Fitness improvement for *Zelda* levels as discussed in Section 6.5

Bibliography

[1] Ian Albert. Video game maps. `http://ian-albert.com/misc/gamemaps.php`, September 2010.

[2] Andrew Apted. *Oblige*. 2010.

[3] Michael J. Apter. A structural-phenomenology of play. In J. H. Kerr and Michael J. Apter, editors, *Adult Play: A Reversal Theory Approach*, pages 18–20. Swets and Zeitlinger, Amsterdam, 1991.

[4] Michael Booth. The AI systems of Left 4 Dead, 2009. `http://www.valvesoftware.com/publications.html`.

[5] Cameron Browne. *Automatic generation and evaluation of recombination games*. PhD in computer science, Queensland University of Technology, Brisbane, Australia, 2008.

[6] Nolan Bushnell, Steve Bristow, and Steve Wozniak. *Breakout*. Atari, 1976.

[7] Mats Carlsson, Nicolas Beldiceanu, and Julien Martin. A geometric constraint over k-dimensional objects and shapes subject to business rules. In *CP*, pages 220–234, 2008.

[8] Earnest Cavalli. GTAIV budget tops gaming records, May 2008.

[9] Carlos A. Coello Coello. Theoretical and numerical constraint-handling techniques used with evolutionary algorithms: a survey of the state of the art. *Computer Methods in Applied Mechanics and Engineering*, 191(11-12):1245–1287, January 2002.

[10] Kate Compton and Matthew Mateas. Procedural level design for platform games. In *2nd Artificial Intelligence and Interactive Digital Entertainment Conference*, pages 109–111, 2006.

[11] Mihaly Csikszentmihalyi. *Flow: The Psychology of Optimal Experience*. Harper Perennial, March 1991.

[12] Steve Dipaola and Liane Gabora. Incorporating characteristics of human creativity into an evolutionary art algorithm. *Genetic Programming and Evolvable Machines*, 10(2):97–110, June 2009.

[13] EA Canada. *NHL 11*. EA Sports, 2010.

[14] Epic Games, Inc. Unreal development kit. `http://www.udk.com/`, November 2010.

[15] Noah Falstein. Understanding fun–the theory of natural funativity. In Steve Rabin, editor, *Introduction to Game Development*, pages 71–98. Charles River Media, Boston, 2005.

[16] Pierre-Alexandre Garneau. Fourteen forms of fun. *Gamasutra. October 12*, October 2001.

[17] Georges Raif Harik. *Learning gene linkage to efficiently solve problems of bounded difficulty using genetic algorithms*. PhD thesis, Ann Arbor, MI, USA, 1997.

[18] Erin J. Hastings, Ratan K. Guha, and Kenneth O. Stanley. Interactive evolution of particle systems for computer graphics and animation. *Trans. Evol. Comp*, 13(2):418–432, 2009.

[19] Rich Hickey. Clojure. `http://www.clojure.org/`, November 2010.

[20] John H. Holland. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, 1975.

[21] R. Hunicke, M. LeBlanc, and R. Zubek. MDA: A formal approach to game design and game research. In *Challenges in Game AI Workshop, Nineteenth National Conference on Artificial Intelligence*, pages 1–5, 2004.

[22] Robin Hunicke and Vernell Chapman. Ai for dynamic difficulty adjustment in games. In *Proceedings of AIIDE 2004*, 2004.

[23] Jesper Juul. Fear of failing? the many meanings of difficulty in video games. In X. Yao, E. Burke, J. A. Lozano, J. Smith, J. J. Merelo-Guervs, J. A. Bullinaria, J. Rowe, P. Tino, A. Kabn, and H.-P. Schwefel, editors, *The Video Game Theory Reader 2*, pages 237–252. Routledge, New York, 2009.

[24] Steven O. Kimbrough, Ming Lu, David Harlan Wood, and Dong-Jun Wu. Exploring a two-market genetic algorithm. In *GECCO '02: Proceedings of the Genetic and Evolutionary Computation Conference*, pages 415–422, San Francisco, CA, USA, 2002. Morgan Kaufmann Publishers Inc.

[25] Raph Koster. *Theory of Fun for Game Design*. Paraglyph Press, November 2004.

[26] Krzysztof Kuchcinski and Radoslaw Szymanek. *JaCoP: Java Constraint Programming Library*. 2010.

[27] Thomas W. Malone. What makes things fun to learn? heuristics for designing instructional computer games. In *SIGSMALL '80: Proceedings of the 3rd ACM SIGSMALL symposium and the first SIGPC symposium on Small systems*, pages 162–169, New York, NY, USA, 1980. ACM Press.

[28] Sid Meier. *Civilization*. MicroProse, 1991.

[29] Shigeru Miyamoto. *Donkey Kong*. Nintendo, 1981.

[30] Shigeru Miyamoto, Shigefumi Hino, and Takashi Tezuka. *Super Mario World*. Nintendo, 1990.

[31] Shigeru Miyamoto, Toshihiko Nakago, and Takashi Tezuka. *The Legend of Zelda*. Nintendo, 1986.

[32] Shigeru Miyamoto, Hiroshi Yamauchi, and Takashi Tezuka. *Super Mario Bros.* Nintendo, 1987.

[33] Newgrounds Inc. *Newgrounds.com–Everything, By Everyone*. 2010.

[34] Tomohiro Nishikado. *Space Invaders*. Midway, 1978.

[35] Takashi Nishiyama and Hiroshi Matsumoto. *Street Fighter*. Capcom, 1987.

[36] Rob Pardo, Jeff Kaplan, and Tom Chilton. *World of Warcraft*. Blizzard Entertainment, 2004.

[37] C. Pedersen, J. Togelius, and G.N. Yannakakis. Modeling player experience in Super Mario Bros. In *IEEE Symposium on Computational Intelligence and Games*, pages 132–139, September 2009.

[38] Paolo Piselli, Mark Claypool, and James Doyle. Relating cognitive models of computer games to user evaluations of entertainment. In *FDG '09: Proceedings of the 4th International Conference on Foundations of Digital Games*, pages 153–160, New York, NY, USA, 2009. ACM.

[39] Chris Remo. MIGS: Far Cry 2's Guay on the importance of procedural content. *Gamasutra*, 11 2008. `http://www.gamasutra.com/php-bin/news_index.php?story=21165`.

[40] Pierre Rivest. *Far Cry 2*. Ubisoft, 2008.

[41] Katie Salen and Eric Zimmerman. *Rules of Play : Game Design Fundamentals*. The MIT Press, October 2003.

[42] Erich Schaefer, David Brevik, Max Schaefer, Eric Sexton, and Kenneth Williams. *Diablo*. Blizzard Entertainment, 1997.

[43] Jurgen Schmidhuber. Developmental robotics, optimal artificial curiosity, creativity, music, and the fine arts. *Connection Science*, 18(2):173–187, June 2006.

[44] Noor Shaker, Georgios Yannakakis, and Julian Togelius. Towards automatic personalized content generation for platform games. In *AIIDE 2010: AI and Interactive Digital Entertainment Conference*. AAAI, 2010.

[45] Karl Sims. Artificial evolution for computer graphics. In *SIGGRAPH '91: Proceedings of the 18th annual conference on Computer graphics and interactive techniques*, volume 25, pages 319–328. ACM Press, July 1991.

[46] Gillian Smith, Mee Cha, and Jim Whitehead. A framework for analysis of 2d platformer levels. In *Sandbox '08: Proceedings of the 2008 ACM SIGGRAPH symposium on Video games*, pages 75–80, New York, NY, USA, 2008. ACM.

[47] Gillian Smith, Mike Treanor, Jim Whitehead, and Michael Mateas. Rhythm-based level generation for 2d platformers. In *FDG '09: Proceedings of the 4th International Conference on Foundations of Digital Games*, pages 175–182, New York, NY, USA, 2009. ACM.

[48] Gillian Smith, Jim Whitehead, and Michael Mateas. Tanagra: a mixed-initiative level design tool. In *FDG '10: Proceedings of the Fifth International Conference on the Foundations of Digital Games*, pages 209–216, New York, NY, USA, 2010. ACM.

[49] Nathan Sorenson and Philippe Pasquier. The evolution of fun: Automatic level design through challenge modeling. In *Proceedings of the First International Conference on Computational Creativity (ICCCX)*, pages 258–267, New York, NY, USA, 2010. ACM Press.

[50] Nathan Sorenson and Philippe Pasquier. Towards a generic framework for automated video game level creation. In *Applications of Evolutionary Computation*, pages 131–140, Berlin, Germany, 2010. Springer.

[51] Penelope Sweetser and Peta Wyeth. Gameflow: a model for evaluating player enjoyment in games. *Comput. Entertain.*, 3(3):3, 2005.

[52] The Freeciv Developers. *Freeciv*. 2010.

[53] The NetHack DevTeam. *Nethack*. 2009.

[54] Julian Togelius and Jürgen Schmidhuber. An experiment in automatic game design. In *IEEE Symposium on Computational Intelligence and Games*, pages 111–118, 2008.

[55] Julian Togelius, Georgios Yannakakis, Kenneth Stanley, and Cameron Browne. Search-based procedural content generation. In Cecilia Di Chio, Stefano Cagnoni, Carlos Cotta, Marc Ebner, Anikó Ekárt, Anna Esparcia-Alcazar, Chi-Keong Goh, Juan Merelo, Ferrante Neri, Mike Preuß, Julian Togelius, and Georgios Yannakakis, editors, *Applications of Evolutionary Computation*, volume 6024 of *Lecture Notes in Computer Science*, chapter 15, pages 141–150. Springer Berlin / Heidelberg, Berlin, Heidelberg, 2010.

[56] Marco Tomassini. Island models. In G. Rozenberg, Th. Bck, J. N. Kok, H. P. Spaink, and A. E. Eiben, editors, *Spatially Structured Evolutionary Algorithms*, Natural Computing Series, pages 11–18. Springer Berlin Heidelberg, 2005.

[57] Michael Toy, Glenn Wichman, Ken Arnold, and Jon Lane. *Rogue*. Artificial Intelligence Design, 1983.

[58] Unity Technologies. Unity: Game development tool. `http://unity3d.com/`, November 2010.

[59] Valve Corporation. *Half-Life*. Sierra Studios and Electronic Arts, 1998.

[60] Georgios Yannakakis and John Hallam. Towards capturing and enhancing entertainment in computer games. *Advances in Artificial Intelligence*, pages 432–442, 2006.

[61] Yerkes, R. M. and Dodson, J. D. The relation of strength of stimulus to rapidity of habit-formation. *Journal of Comparative Neurology and Psychology*, 18:459–482, 1908.

[62] Derek Yu. *Spelunky*. Independently Published, 2008.